Poly-algorithmic Techniques in Real Quantifier Elimination

Tonks, Zak

Award date:
2021

Awarding institution:
University of Bath

Poly-algorithmic Techniques in Real Quantifier Elimination

submitted by

Zak Tonks

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

January 2021

Summary

Quantifier Elimination over the Reals (QE) is a topic in Computer Algebra concerning elimination of quantifiers from boolean formulae of polynomial constraints. QE is known to be worst case doubly exponential time complexity in the number of variables, so optimisation in this area is always of interest. Otherwise, improvements with respect to interface can make usage of QE more tractable. QE problems arise in a range of contexts, notably motion planning (such as robotics), biology, mechanics, and finance.

Largely, this work discusses the implementation of a new package `QuantifierElimination` for the Computer Algebra software Maple. This package, developed in collaboration with Maplesoft, is an amalgamation of contemporary research on two main algorithms for QE, Virtual Term Substitution (VTS) and Cylindrical Algebraic Decomposition (CAD), but a main focus is an investigation into the efficiency & efficacy of a new "poly-algorithm" between these two algorithms.

The implementation of CAD includes the first known usage of the Lazard projection for CAD in Maple, and the first known implementation & investigation of new research for equational constraint optimisations with the Lazard projection in CAD. Such equational constraint optimisations may induce mathematical "curtain" occurrences which are ordinarily cause for failure to construct a CAD with frequently desired properties. In conjunction with the first implementation of relevant algorithms from other research, methodology is delineated to ignore, or else attempt to recover from *any* general failure to construct or evaluate parts of the CAD for QE, including curtains. `QuantifierElimination` also allows for generation of full CADs without the context of QE, where users can inspect individual cells via various object methods, with the package being largely object oriented where appropriate. Generation of CADs in unquantified contexts has uses in motion planning and real algebraic geometry. Quite often the thesis focuses on finer implementation details & challenges, including with respect to the recent research on equational constraints.

Other aims for the package are rich output and "incrementality", to meet the usual aims of implementations of algorithms for QE to be used within Satisfiability Modulo Theory (SMT) solving for the theory of real arithmetic (`QF_NRA`). Rich output includes production of "witnesses" for fully homogeneously quantified problems, which prove the equivalence of a subset of problems to their quantifier free equivalent. Extension of existing methods enables production of witnesses where the framework of the new poly-algorithm is concerned. The poly-algorithmic methods are extended to be incremental for QE as well. There are a wealth of keyword options and customizable levels of user information to print for users, as the package aims to reconcile with Maple's status as a mathematical toolbox to be easy to use with in depth information being readily available for experienced geometers. Other functions are more pedagogical to accommodate users new to problem formulations in QE and real algebraic geometry.

The software is compared via benchmarking against existing QE and CAD software, especially existing packages in Maple, and investigates case studies on examples to demonstrate new methodologies and research.

Acknowledgements

The largest thanks goes to my supervisor James Davenport. It goes without saying that I couldn't have done this without you. Despite being spread thin amongst so many exploits in Computer Science, Computing, and Mathematics, your passion shines through in each. I'm grateful for your support and belief in me throughout, and appreciate that you extended this opportunity to me. As a whole, this project allowed me to visit a great many places — this can't be said for many occupations.

I thank Maplesoft for their immense support & hospitality throughout the project. I felt incredibly welcome at the offices and surroundings in Waterloo, Ontario. Being a Mathematics graduate, I was largely underinformed as to the scope of good software development practices, and computing in general. My absolute special thanks to both Jürgen Gerhard & Stephen Forrest, who's Maple and programming expertise was invaluable. I greatly enjoyed being able to see near enough all of Ontario's sights thanks to Jürgen. Maplesoft make amazing software to do mathematics with. It's been a pleasure contributing to Maple, and I hope I can do so again.

This research was funded by the EPSRC under grant reference EP/N509589/1 with additional support from Maplesoft. Bath, and the University of Bath, have essentially been my home for ≈ 8 years at the conclusion of this project, and in particular two academic degrees. It's been a pleasure to have been able to learn and teach in these historic surroundings for so long. It's incredible that the university manages to attract so many hard working individuals with exceptional integrity and kindness, and I'm grateful for many at Bath who've made the experience so enjoyable. Friends of many years all originating from the realm of Mechanical Engineering at Bath, especially Simon & Lewis, all deserve my utmost gratitude. My friend Jonny also deserves special thanks for his support.

Matthew England (University of Coventry) deserves thanks for his constantly useful discussion & interest in my work. His dedication to research in Computer Algebra is clear and I hope it continues. Gregory Sankaran (University of Bath) provided much encouragement in the (very!) early days of my research, and is a lot of the reason for my interest & early knowledge of real algebra.

Ben Pring (previously University of Bath, now University of South Florida) has made for a great source of support. I learned a great deal from his research experience and supporting comments starting from the very early days when I was a research intern. He deserves nothing but the very best in his career going forward.

Thanks to Casey Mulligan (previously University of Chicago) for contributing the economics QE problems that make for part of the QE database for this project.

Fabrice Rouillier (INRIA) deserves thanks for being amenable to further development of low level real root isolation & refinement procedures to later improve `QuantifierElimination`'s CAD implementation. Hopefully he and his colleagues find his suggested features of `QuantifierElimination`'s CAD useful or interesting for their work & research.

Various conferences & workshops in Computer Algebra have been enjoyable and provided essential discourse for this work, and I appreciate various individuals who have made enumerable trips abroad more enjoyable.

Many thanks to my examiners Gregory Sankaran and Thomas Sturm, with special thanks to Thomas for his interest in the software and many very interesting comments. I especially appreciate the effort in reading such lengthy work.

Special thanks to my parents Mark & Yvette. Their support has been interminable, and I owe an indescribable amount to them for me to be where I am.

Contents

List of Algorithms

List of Figures

Notation

Maple Packages

Many Maple packages are referred to throughout - they are usually typeset as e.g. `QuantifierElimination`, where "QuantifierElimination" is the package in question. Further, this extends to other software packages, such as typesetting the software "QEPCAD B" as `QEPCAD B`.

Read QuantifierElimination:-QuantifierEliminate as "the function QuantifierEliminate within the Maple package `QuantifierElimination`". The infix `:-` delimits this notion.

The Real Numbers, Real Algebraic Numbers & Functions

Throughout, $\mathbb{R}$ is identified with the countable set of real algebraic numbers (Definition 30) to reconcile with the interest in real closed fields (Chapter 1). As such, an "irrational number" here is always algebraic, rather than a truly transcendental real number such as π. If $p \in \mathbb{R}[x]$, and $a, b \in \mathbb{Q}$, $a < b$, then read $\mathrm{RootOf}(p, a..b)$ as "the real root of the polynomial p lying in the open interval (a, b)" (where and when such a real root exists). Such an expression is actually valid as a construct in Maple, and the construction essentially completely coincides with the definition of a real algebraic number (Definition 30), where the root is irrational. The term "RootOf" may be used to mean "strictly irrational real algebraic number", i.e. a number in $\mathbb{R} \setminus \mathbb{Q}$, as a rational number needn't be expressed by a RootOf, nor would Maple accept the expression $\mathrm{RootOf}(x - a, a..a)$ $(a \in \mathbb{Q})$ under evaluation. Note that ".." is the Maple infix operator for an interval.

An expression $\mathrm{RootOf}(p, \mathrm{index} = \mathrm{real}[i])$ where $p \in \mathbb{R}[\_Z, y_1, \ldots, y_k]$, $i, k \in \mathbb{N}$, $\deg(p) > 0$ is a "parametric RootOf", or a "real algebraic function" (defined formally as Definition 31). Again this is a valid construct in Maple, and the indexing "index = real[i]" implies the knowledge that p has a real root under valid constraints on $y_1, \ldots, y_k$. Maple uses $\_Z$ as the universal dummy variable to denote the variable that the RootOf is with respect to. This is true of real algebraic numbers under evaluation in Maple, where $\mathrm{RootOf}(x^2 - 2, 1..2) \to \mathrm{RootOf}(\_Z^2 - 2, 1..2)$ under Maple evaluation. i is certainly such that $i \leq \deg_{\_Z}(p)$, and the indexing is in terms of increasing real values, i.e. $\mathrm{RootOf}(p, \mathrm{index} = \mathrm{real}[1]) < \mathrm{RootOf}(p, \mathrm{index} = \mathrm{real}[2]) < \ldots$. Again, the assertion is that such a real algebraic function only appears with assertions

on $y_1, \ldots, y_k$ such that the root exists for those values of $y_1, \ldots, y_k$ and is real. A real algebraic function is not equivalent to a real algebraic number unless or until the expression is evaluated at real numbers for $y_1, \ldots, y_k$. In Maple, RootOf initialisation is such that $\text{RootOf}(x^2+y^2-2, x, \text{index} = \text{real}[1]) \to \text{RootOf}(\_Z^2+y^2-2, \text{index} = \text{real}[1])$, i.e. one specifies the variable with which to replace with the dummy variable $\_Z$. Because $p \in \mathbb{R}[\_Z, y_1, \ldots, y_k]$, the representation of the real number coefficients of p in Maple may be real algebraic numbers represented by RootOfs indexed by intervals as shown in the paragraph above.

Objects

Much of the Maple package to be described is programmed in an object oriented way, hence the algorithms detailed within often refer to objects and their members. If "QEClass" is a class of objects, then it will be typeset as `QEClass`. If "Obj" is an instance of a class, and "prop" one of its "members" or "properties", then read $\text{Obj} \mapsto \text{prop}$ as "the prop member of Obj". Most often algorithms will use this typesetting.

Substitution

Take $\Psi[x \,/\, t]$ to mean "the algebraic substitution of t as the variable x in Φ", where t is an algebraic number. This contrasts with $\Psi[x \,//\, t]$, which is defined as the *virtual* substitution of t for x in Φ (Definition 7), where t is a virtual substitution term.

Miscellaneous

We take $\mathbb{N}$ as "the natural numbers starting at 1", hence $1, 2, 3, \ldots$. Meanwhile $\mathbb{N}_0 \coloneqq \mathbb{N} \cup \{0\}$.

If A is a container type data structure, take $|A|$ to be the total number of elements of that container. We may occasionally use negative indexing for container type data structures, i.e. $A[-i]$ is the ith element counting from the end of the container, for suitable $i \in \mathbb{N}$, $i \leq |A|$. For example, $A[-1]$ is the last element of the container A.

Other programming notation familiar from languages such as `C` may be used, such as i`++` to mean post-increment i, i.e. "use the value of i, then increment". Similarly i`--` is post-decrement, i.e. "use the value of i, then decrement". Meanwhile `++`i and `--`i mean pre-increment and pre-decrement respectively — "increment/decrement, then use the resulting value of i".

Read $i \leftarrow x$ to mean "Assign the value x to i". This is usually used in the typesetting of algorithms. Similarly $i \coloneqq x$ means assignment of the value x to i, but in a Maple context. The semi colon "`;`" in an algorithmic or programmatical context is a statement separator.

There is much discussion of objects that form a typical tree structure. These trees are viewed to grow "root down", with the root node at the top of the tree.

Chapter 1

Introduction to Quantifier Elimination over the Reals

Throughout, we are concerned with algorithms for providing solutions for "Quantifier Elimination" over real closed fields (henceforth shortened to "QE"). QE is a powerful tool offering concise symbolic solutions for a wide range of problems across mathematics, but its use is impeded by the significant complexity of the associated algorithms. Much research focuses on the optimisation of such algorithms, and otherwise providing intuitive input and rich and meaningful output alongside QE. The research of this work also focuses on these aims. We precisely define necessary terms needed for QE.

Definition 1 (Atom). *An* atom *is a quantifier free polynomial constraint (an object of the form* $f \rho\ 0$ *where* $f \in \mathbb{Z}[x_1, \ldots, x_n]$*,* $n \in \mathbb{N}$*,* $\rho \in \{=, \leq, \geq, <, >, \neq\}$*),* true*, or* false*.*

Definition 2 (Quantifier). *A quantifier symbol is either* $\forall$ *or* $\exists$*. A quantifier is a quantified symbol followed by a variable, e.g.* $\exists x$*. A quantified variable is any variable that appears in a quantifier for a QE problem. A block of quantifiers* $\exists x_1, \ldots, \exists x_n$ *may be formatted as* $\exists \mathbf{x}$*.*

Definition 3 (Tarski Formula). *A* Tarski Formula *(TF) is a polynomial constraint, i.e.* $f \rho\ 0$ *where* $f \in \mathbb{Z}[x_1, \ldots, x_n]$*,* $n \in \mathbb{N}$*,* $\rho \in \{<, \leq, \neq, =\}$ *or a boolean combination of Tarski formulae, where allowable boolean operators may feature* $\wedge, \vee, \Rightarrow, \veebar$*.*

A "Quantified Tarski Formula" is a Tarski formula where any subformula, or indeed the whole formula may feature quantifiers within. A Quantifier Free Formula (QFF) is a Tarski formula without quantifiers (additionally take QF to mean "quantifier free").

The assertion that $\rho \in \{<, \leq, \neq, =\}$ is purely due to Maple's convention to use $<$ and $\leq$ instead of $>$ and $\geq$ (without loss of generality, this is all we need, as e.g. $f < g \equiv g > f$). $f \in \mathbb{Z}[x_1, \ldots, x_n]$ can be replaced by $f \in \mathbb{Q}[x_1, \ldots, x_n]$ in the above definitions, because there is a trivial conversion from relations on rational polynomials to relations on integral constraints by multiplying through by denominators. Again, without loss of generality, it suffices to consider the polynomials as integral in Tarski

formulae. The above can be expressed in a more canonical form amenable to discussion and computation — any such formula can be converted into the form below, and as such further from this section we will always assume top level input is "prenex" without loss of generality.

Definition 4 (Prenex Quantified Tarski Formula). *An expression*

$$Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi(x_1,\dots,x_n) \tag{1.1}$$

where $Q_i \in \{\forall, \exists\}$, $i = n-m+1,\dots,n$, $1 \leq m \leq n$, *and* Φ *is a quantifier free Tarski formula that only features the boolean operators* $\wedge$ *or* $\vee$, *is a* Prenex Quantified Tarski Formula. $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n$ *is called the* prefix. *The variables* $x_1,\dots,x_{n-m+1}$, *i.e. those appearing in* Φ *but not the prefix, may be referred to as "free" or unquantified variables.*

While the intention of "prenex form" is not only to force all quantifiers to manifest at the beginning of an expression, it also ensures there are no conflicts between variables such as the expression $\forall x\ P(x) \wedge \forall x\ Q(x)$, which is actually equivalent to $\forall x \forall x_1\ P(x) \wedge Q(x_1)$ via prenex conversion, but more particularly the "alpha conversion" of one instance of x to a new renaming, say x_1. Note that many QE problems will not manifest in prenex form — that is, it is more likely that they manifest in such a way that quantifiers exist within boolean operators, and further boolean operators are present such as $\Rightarrow, \veebar, \neg$. However we note that there always exists a conversion of such problems to prenex form, by using the variable renamings i.e. alpha conversion where necessary, or using logical equivalences e.g. $a \Rightarrow b \equiv \neg a \vee b$. Then, without loss of generality we can always work with prenex formulae, and do so henceforth. In particular the number of quantified and total variables m and n for a problem are always the numbers of variables as of conversion to prenex form, and in particular alpha conversion — because a prenex formula asserts that all quantifiers must precede a quantifier free formula, variable conflicts can no longer arise.

ConvertToPrenexForm from the `QuantifierTools` subpackage of `QuantifierElimination` (Chapter 6) provides prenex conversion for general formulae that may be non prenex. AlphaConvert provides alpha conversion, renaming variables in conflicts without the moving of quantifiers, such that $\forall x\ P(x) \wedge \forall x\ Q(x) \mapsto \forall x\ P(x) \wedge \forall x_1\ Q(x_1)$. Formulae passed to top level QE functions in `QuantifierElimination` are converted to prenex form "under the hood" before QE computation. The notion of "prenex" in terms of quantifiers preceding a quantifier free formula applies to further extensions on Tarski formulae to be defined in due course.

Definition 5 (Quantifier Elimination). *Quantifier Elimination (QE) is the problem of eliminating all quantifiers* $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n$ *from (1.1) to receive an unquantified boolean formula of relations in* $x_1,\dots,x_{n-m}$, *or* true, *or* false. *When* $n = m$, *there are no free variables and the formula is "fully quantified", so (1.1) is certainly equivalent to* true *or* false.

Furthermore, `QuantifierElimination` accepts even more general formulae than non prenex Tarski formulae. QE by CAD can accept a modification of Tarski formulae

allowing for general real polynomials, as opposed to merely integral (Definition 32). The discussion on such formulae remains in Chapter 3 as the representation of irrational real algebraic numbers within those formulae is relevant. We importantly note that $\mathbb{R}$ is identified with the set of real algebraic numbers as given by Definition 30. Even further, we can accept rational functions within input for QE.

Definition 6 (Rational Tarski Formula). *A* Rational Tarski Formula *is a constraint on a rational function, i.e.* $r \rho\ 0$ *where* r *is a rational function* $r = \frac{f}{g}$ *where* $f, g \in \mathbb{R}[x_1, \ldots, x_n]$*,* $n \in \mathbb{N}$*,* $\rho \in \{<, \leq, \neq, =\}$ *or a boolean formula of Rational Tarski formulae, where allowable boolean operators may feature* $\wedge, \vee, \Rightarrow, \veebar$*.*

ConvertRationalConstraintsToTarski from the `QuantifierTools` package provides conversion from applicable rational Tarski formulae to real Tarski formulae (Definition 32 — constraints with real algebraic numbers are allowable in a rational Tarski formula). Again, top level QE functions in `QuantifierElimination` can do this conversion "under the hood". The conversion uses facts such as $\exists x \exists y\ \frac{x}{y} = 0 \equiv \exists x \exists y\ x = 0 \wedge y \neq 0$, $\exists x \exists y\ \frac{x}{y} > 0 \equiv \exists x \exists y\ xy > 0$, and similar facts for other operators obtained similarly. Note that existential quantification is generally necessary for such facts. Further, we still assume that input formulae where VTS is concerned are prenex Tarski formulae, and where CAD is concerned, prenex real Tarski formulae.

Tarski showed that computing QE over the reals is decidable in 1951 [62], however his associated scheme was essentially infeasible to implement. In 1975, Collins [18] showed that QE via Cylindrical Algebraic Decomposition is at most worst case doubly exponential time complexity (in the number of variables), but in 1988 Davenport and Heintz [22] constructed examples that prove that QE can inherently be at least worst case doubly exponential as well. Nowadays, two of the most well known algorithms to achieve QE are Virtual Term Substitution (VTS, Chapter 2) and Cylindrical Algebraic Decomposition (CAD, Chapter 3). This project concerns and implements both, and a poly-algorithm between the two. Largely, most current implementations of QE systems use CAD, as it can cope with input formulae containing polynomials of *any* degree. In contrast, VTS is largely only feasible for low degree problems. We acknowledge other approaches to QE involving other algorithms such as Fourier-Motzkin elimination [52] (much like VTS, this is only applicable for low degree problems), but only focus on VTS and CAD in terms of explicit QE algorithms in this work.

1.1 Applications of Quantifier Elimination

Nonlinear optimisation problems and (conditions on) solvability of systems of multivariate polynomial equations and inequalities can canonically be presented as QE problems. QE problems are known to arise in economics [54], mechanics [39], mathematical biology [15, 11], AI to pass exams [68], and motion planning [71].

1.1.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is a variant of the boolean satisfiability (SAT) problem that allows for first order logic to appear within atoms. As usual, the aim is to check whether a given formula is satisfiable (SAT) or unsatisfiable (UNSAT). In particular, the "theories" defining the logic relevant to this work are `QF_NRA` — "quantifier free non linear real arithmetic" and `QF_LRA` — "quantifier free linear real arithmetic". In particular there is a focus on `QF_NRA` in the context of SMT for this work. These are formulae that as unquantified formulae reconcile with those as defined by Definition 3, however SMT formulae are certainly in conjunctive normal form (CNF), and are written as unquantified, even though one should view a `QF_NRA` SMT formula as a fully existentially quantified Tarski formula

$$\exists x_1, \ldots, \exists x_n \ \Phi(x_1, \ldots, x_n)$$

where Φ is in conjunctive normal form (CNF). However, this representation does obscure some of the differences in the *methodology* between SMT and QE, where SMT largely acts incrementally on clauses, using other bespoke techniques such as backtracking inherited from SAT methodology. In particular for `QF_NRA`, QE tools such as VTS and CAD are often deployed beneath SMT solvers in an incremental fashion amongst the clauses of the formula for SMT. Full incrementality is one of the major features in `QuantifierElimination` for all the available QE methodologies, such that it can be used by an SMT solver. Additionally, `QuantifierElimination` supports production of witnesses (Sections 2.5, 3.10 and 4.4.1) in order to prove the satisfiability of fully existential quantified formulae, or analogously the unsatisfiability of fully universally quantified formulae. [1] discusses that neither of the QE algorithms (or combinations thereof) focused on in this project provide good "proofs" of the reverse cases, i.e. unsatisfiability of an existentially quantified formula or satisfiability of a universally quantified formula. A previous example of VTS and/or CAD being used in an incremental manner for SMT solving is `SMT-RAT` [19].

1.2 Maple and `QuantifierElimination`

Maple is Computer Algebra software produced by Maplesoft, the first version of which was released in 1982. Its use as a Mathematical toolbox has seen it popular in education and industry alike, with the most recent version as of the time of writing being Maple 2020. `QuantifierElimination` is a package written in Maple in collaboration with Maplesoft for this project, and is the culmination of this work in terms of usable software. A software demo of `QuantifierElimination` can be found at [67]. Given the collaboration, there is a focus on good coding practices with respect to Maple. There is a focus on mutable data structures to avoid the cost of garbage collection in building objects via a series of intermediate immutable data structures, such as lists, which are the rudimentary data structure for storage in Maple. Many parts of the implementation of `QuantifierElimination` are object oriented to enable "methods" for incrementality, but also such that data structures returned to users in an

outward facing sense are intelligible as objects, with features such as bespoke typesetting. `QuantifierElimination` features the subpackage `QuantifierTools` (Chapter 6) to enable further understanding of QE and related concepts such as "Tarski-like" formulae. Via a focus on rich user output to reconcile with Maple's status as a mathematical toolbox, `QuantifierElimination` provides features extending on QE, such as providing the capability of generating full CADs without the context of QE, "witnesses" for homogeneous QE problems (Sections 2.5, 3.10 and 4.4.1), and general "evolutionary" methods (Chapter 5) with a view to assisting with deeper understanding of QE problems. "Evolutionary" is terminology that would usually broadly be referred to as "incremental" in other QE literature. The most notable functions of `QuantifierElimination` are QuantifierEliminate, using the poly-algorithm described in Chapter 4, PartialCylindricalAlgebraicDecompose, performing QE purely by Partial CAD (Section 3.4), and CylindricalAlgebraicDecompose, creating general full CADs on formulae or sets of polynomials without evaluation of truth values. The CAD functions of `QuantifierElimination` are not to be confused with those below `RegularChains`' subpackage `SemiAlgebraicSetTools` — in particular the functions named PartialCylindricalAlgebraicDecompose actually have fairly distinct purposes. The implementation of CAD in `QuantifierElimination` is a projection and lifting CAD using the Lazard projection.

Much of the thesis discusses the detail behind the implementational aspects of `QuantifierElimination`, in particular delineating lower level algorithms behind various features, and discussing challenges of implementation of various features. This is especially with respect to equational constraint optimisations for the Lazard projection in CAD, and very recent research moving these optimisations towards being complete (Section 3.7.2).

1.3 Other QE Related Software

Acknowledgement of other implementations of QE, including in other Computer Algebra systems is necessary, as the other implementations are often compared to through the course of this work, especially in benchmarking `QuantifierElimination` against a subset of them in Chapter 7.

- `QEPCAD B` [8] is an implementation of CAD for QE written in `C`, using the public `SACLIB` Computer Algebra library. In particular, `QEPCAD` was written to demonstrate the first usage of the "Partial CAD" methodology, an optimisation of CAD for QE (Section 3.4), and `QEPCAD B` is a more recent branch.
- `Redlog` [24] for the Computer Algebra system `REDUCE` features implementations of VTS and CAD to achieve QE. The function rlqe performs general QE by VTS, switching to CAD upon intermediate formulae exceeding degree 2 in further quantified variables (a degree violation), or rlcad performs QE solely by Partial CAD. (Hence rlqe and rlcad are somewhat analogous to QuantifierEliminate and PartialCylindricalAlgebraicDecompose from `QuantifierElimination`). `Redlog`'s rlgcad is a function implementing a far more general version of what would be OpenCAD (Section 3.5) in this work.

- `SyNRAC` [74] is a package publicly available for Maple that implements VTS and CAD for QE. Similarly to `Redlog`, SyNRAC:-qe performs general QE by VTS, switching to CAD upon degree violation, which again for SyNRAC is degree 2.

- `RegularChains` [3] (RC) is a package included with Maple by default, with the latest versions of the package and source code also available at their site. It includes an implementation of CAD, but unlike most implementations of CAD it is not a projection & lifting CAD. It works via triangular decompositions to create a CAD in complex space before conversion to a CAD in real space. It is fully incremental [17]. As of Maple 2020.1, `RegularChains` provides QE via its CAD with RegularChains:-SemiAlgebraicSetTools:-QuantifierElimination [47], and CAD outside of the context of QE is provided by RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose. Various auxiliary tools for producing expressions necessary to use the top level functions are also part of the `RegularChains` module, such as PolynomialRing.

- `ProjectionCAD` [27] is a package publicly available for Maple providing projection & lifting CADs using technologies provided by `RegularChains`. Unlike the other listed packages, it does not inherently support QE, but investigates several CAD optimisations of interest, such as equational constraints, and offers support for variable strategy in terms of metrics on generated projection bases. The projection operator is McCallum's, as was contemporary at the package's time of writing.

- `SMT-RAT` [19] is a toolbox implementing VTS and CAD with incrementality in order to provide an SMT solver for the theory of real arithmetic. It is written in `C++`.

- `RealPolynomialSystems` [61] in the Computer Algebra system Mathematica. The function Reduce offers Quantifier Elimination over the Reals (and other domains). For the real numbers, it offers VTS up to a degree violation of 2 (viewed as a preprocessing step), and as usual offers CAD beyond that, with equational constraints and Gröbner bases offered as options for optimisation of such.

Chapter 2

Virtual Term Substitution

Virtual Term Substitution (henceforth VTS) was first defined by Weispfenning in 1988 [69]. This provided elimination of variables appearing linearly, but later in 1997 he extended this to the case for variables appearing quadratically [70]. The methodology revolves around substitution of terms for quantified variables to attempt to find examples or counterexamples for existential or universal quantifiers respectively. As of the point of writing, the univariate case was well studied, with several improvements owing to Košta [43]. Indeed, the implementation of VTS in `QuantifierElimination` uses many algorithms suggested by Košta. We detail the mechanisms behind VTS to better understand the multivariate case, which is the backbone behind QuantifierEliminate.

2.1 Background

Virtual Term Substitution works to eliminate quantifiers where the corresponding variables appear as low degree in the polynomials from $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi$. The formula Φ must be integral, i.e. the polynomials in the constraints must be over $\mathbb{Z}$ and so $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi$ must reconcile with Definition 3. In reality, $\mathbb{Q}$ will do, as any formula over $\mathbb{Q}$ is equivalent to one over $\mathbb{Z}$. When a formula passed to QuantifierEliminate is not integral, the formula is instead passed directly through to QE by Partial CAD (Chapter 3), because CAD can deal with general real polynomials, but VTS cannot. The user is notified that this is the case when applicable. Note that poly-algorithmic QE (Chapter 4) is inapplicable in a non trivial sense in this case too.

Most of the constituent parts of VTS in `QuantifierElimination`, especially those with respect to VTS to eliminate one variable, are based off of Košta's thesis [43]. `QuantifierElimination` implements functions with (essentially) identical names as those formulated there to achieve essentially always the same purpose. These include functions such as atposl, PC-to-TPs, at-cs, at-cs-fac, expand-eps-at, vs-inf-at, pseudo-sgn-rem, vs-prd-at, vs-at, and guard. One notes that certain optimisations such as prime constituents, conflation, and degree shift from [43] are not implemented. Clustering of linear or quadratic test points is implemented, and usage of which is controlled by the keyword option `'UseClustering'` in top level QE functions. One can control

the choice of "bounds" for VTS to use by the keyword option `'bounds'`, which takes a non empty set up to and including the symbols `'lower'` and `'upper'`.

VTS is only applicable to eliminate variables such that they appear in polynomials with degree such that appropriate methodology for substitution has been described for that degree. Linear and quadratic elimination owe to the initial presentations by Weispfenning [69, 70], and [43] delineated elimination of variables appearing cubically. In particular, all constraints in the formula must be such that the associated polynomials *factor* to polynomials of at most degree 3, i.e. the limitations on degree really apply only to irreducibles. Due to the limitations on VTS in terms of degree, some software and/or researchers view VTS as more of a pre-processing step before any other complete algorithm for QE such as CAD (as is the case in documentation for `RealPolynomialSystems` in Mathematica [61]). `QuantifierElimination` implements the case up to and including quadratic VTS, with support for cubic elimination in development, using much the same technology as that of any other degree case, with most of the work for extension to another degree being enumeration of polynomial real types and formulae schemes. One notes that degree 4 VTS was one of the suggestions for further work of [43], and the current degree limitations on VTS are a reflection of the current work expended in research to provide methodology up to degree 3 thus far. As such, the sentiment of VTS only being applicable to low degree situations is a practical rather than theoretical impediment, although the effort required to generate the associated schemes for elimination increases commensurate with degree due to the increasing numbers of polynomial real types. In `QuantifierElimination`, the keyword option `'MaxVSDegree'` takes any value up to 2 to control the maximum degree that VTS will use for elimination.

We provide an overview as to the mechanism of VTS in terms of [43] and how various functions there are used within `QuantifierElimination`. In particular, we discuss the case for elimination of a single quantifier $\exists x\ \Phi(x, \mathbf{u})$ where u is a list of "parameters", i.e. free variables for the formula. As such, in terms of Computer Algebra, VTS truly views polynomials within Φ as represented "recursively", with the main variable being x, but the ordering of variables in $\mathbf{u}$ being largely superfluous.

Virtual Term Substitution revolves around "test points" generated from a quantified formula which describe the real root of a (potentially multivariate) polynomial in a particular variable. Generation of test points first requires usage of atposl to generate "candidate solutions" from every relation of the quantified formula, in terms of the relational operator and polynomial of the constraint. Because `QuantifierElimination` does not implement prime constituents, atposl is the only generator of candidate solutions, and is done entirely recursively to every atom of the formula to generate one set. at-cs is the function providing candidate solutions for a single atom, and these candidate solutions in `QuantifierElimination` are Maple lists almost identical in structure to those formatted in the original presentation. The set of candidate solutions is then processed by PC-to-TPs to produce a set of structural test points associated to the formula, which takes into account the value of the key word `'bounds'`. Such structural test points may represent the exact substitution of the real root of some polynomial.

Alternatively, because of the presence of strong relations where $\rho \in \{<, \neq\}$, these structural test points may feature the infinitesimal ∞ or ε. Where a test point features $\pm\infty$, the test point entirely represents the virtual substitution of an "extremal" value — a value exceeded by or exceeding the values of all real roots of all polynomials from the expression. Test points featuring ∞ are not associated to any particular polynomial. Where ε is concerned, the root description of a polynomial must be provided, and the test point implicitly means "substitution of a value just less than $(-\varepsilon)$/more than $(+\varepsilon)$ this real root". Usage of infinitesimals is centric to virtual substitution, and is some of the meaning behind the word "virtual". The remainder of the intention behind the terminology "virtual" is because of substitution of real roots of polynomials with respect to one variable into other polynomials that are potentially multivariate.

The intention of generation of structural test points is such that there is at least one structural test point from every interval of the real line formed by real roots of the polynomials of the processed formula, and the generated set of test points is finite. Usage of the infinitesimals ∞ and ε assist with this by allowing for description of points from within "open" intervals. Because the polynomials are potentially multivariate, the generation of test points is via all possible real types for the associated polynomials. The "real type" of a polynomial informally describes its "shape" in terms of the number of real roots it has, and more precisely the sequence of signs its values take when moving across the real line. As an example, a "positive" quadratic donating just one real root has real type 2, with sign sequence $(1, 0, 1)$, and a negative parabola donating two real roots has real type -1, with sign sequence $(-1, 0, 1, 0, -1)$.

Structural test points in `QuantifierElimination` are represented by Maple lists essentially encapsulating the information prescribed by (f, S) in Košta's vs-at. This typically includes a polynomial f, variable for substitution, an infinitesimal $\pm\varepsilon$ or 0, assumed degree d of f, real type of f, and the index of the real root the test point represents. d should be such that $0 < d \leq \deg_x(f)$ — a programmatic notification that we are actually considering substitution from an appropriate reductum of f when the relevant coefficients are able to vanish (which is a fact in itself dealt with by guards). The special virtual substitution terms for $\pm\infty$ are $[0, x, -\infty]$ and $[0, x, \infty]$, i.e. they truncate due to not representing roots of any real polynomial. Therefore the function subsituteWholeExpression imitating Košta's vs-at and implementing virtual substitution in terms of Definition 7 takes a relation and a structural test point represented as such as list.

When substituting a root of f, where f is of degree d in x into $g \,\rho\, 0$, considering at a root of f, $f = 0$, it suffices to substitute the root of f into $h \coloneqq pseudo - sgn - rem(g, f, x)$. pseudo-sgn-rem is a function defined in [43], which is a variant of pseudoremainder (further prem) preserving sign conditions on g after pseudoremainder via any assignment of the parametric variables. Via properties of pseudoremainder, h is of degree at most $d-1$, and it suffices to know how to substitute a root of a polynomial of degree d into a polynomial of degree $d - 1$ or less. This knowledge is provided by Košta's "formula schemes", which essentially provide a lookup table to provide results of virtual substitution of structural test points of degree d into any nontrivial degree relation less than d, $h \,\rho\, 0$. `QuantifierElimination`'s function formulaScheme implements this lookup table returning the relevant Tarski formula in terms of the

input relation and data from a structural test point. If h is of degree at most 0, then we can return $h\,\rho\ 0$ as the result of $g\,\rho\ 0[x\,//\,T]$, where T is the structural test point associating information about substitution of the real root of f.

Another element of virtual substitution is "guards". Guards define a map from structural VTS test points to Tarski formulae, where a guard represents the conditions that must be valid such that substitution of a structural test point would be valid. This offers credibility to usage of substitution points from multivariate polynomials that may otherwise vary in the number of real roots presented with respect to any one variable — if one is to assert various conditions, then one is able to use the substitution. As an example, substitution from a linear polynomial in x requires that the coefficient of x is non zero, such that the polynomial is genuinely a polynomial in x at all. Further, substitution of the second distinct root in x of a quadratic requires assertions that the discriminant in x is strictly positive. guard in `QuantifierElimination` generates such guards from structural test points, and as usual the function is essentially a lookup table formed from [43, Section 2.5.2]. Hence, virtual substitution of a test point into any formula is in some sense "predicated" on the result of its guard, and the guard for the test point must be conjuncted with the result of virtual substitution. Because a structural test point may consider substitution from a reductum of a polynomial, the function guard includes the conditions that the relevant leading coefficients must be 0 such that the polynomial would be that degree when generating the guard for such a test point.

Definition 7 (Virtual Substitution). *[43, Section 2.3] The virtual substitution of a structural test point T for x into a quantifier free relation $g(x, \mathbf{u})\,\rho\ 0$ is $F(\mathbf{u}) \coloneqq g[x\,//\,T]$, a quantifier free formula such that for any parameter values $\mathbf{a} \in \mathbb{R}^{n-1}$, if $\mathbf{a}$ satisfies the guard of T, then $\mathbf{a}$ satisfies F if and only if $\mathbb{R} \models (g\,\rho\ 0)(\mathbf{a}, T[\mathbf{u}\,/\,\mathbf{a}])$.*

If Ψ is non atomic, the call vs-at(Ψ, T, x) where the algorithm vs-at is defined in [43] defines $\Psi[x\,//\,T]$, and is recursive to the atoms of Ψ.

We discuss how to substitute test points involving infinitesimals. In the case of infinity, vs-inf-at describes the formula resulting from substitution of "$x = \pm\infty$" into an atom. vs-inf-at is recursive on the (monotonically decreasing) degree of the atom to act upon, and returns a Tarski formula. Hence, in this case, every atom of the formula to act upon with respect to virtual substitution is replaced by the results of vs-inf-at. In the case of ε, our virtual term for substitution is $x = t \pm \varepsilon$ for t some real root of a polynomial, i.e. a regular virtual substitution term. Again, virtual substitution recurses to the leaves of the formula to act upon in this case, and vs-at has that we use expand-eps-at on an atom, which produces a Tarski formula again via recursion on the monotonically decreasing degree of the relation. Having "epsilon expanded" the atom, we can now substitute the structural test point corresponding to "$x = t$", i.e. the test point sans any infinitesimals into the results of epsilon expansion. One can view the epsilon expansion via expand-eps-at as preprocessing the relation such that we can consider the case for a point "just to the left/right" of t.

In total, virtual substitution works to find valid "examples" to eliminate an existentially quantified x. For the structural test point elimination set generated from the

atoms of a formula $\exists x\ \Phi$, $t_1, \ldots, t_k$, $k > 0$, VTS eliminates $\exists x$ via the formula

$$\bigvee_{i=1}^{k} G(t_i) \wedge \Phi[x \,//\, t_i], \tag{2.1}$$

where G is the map inherited from generation of guards, and virtual substitution as defined by Definition 7.

VTS acts entirely on one quantified variable at a time, in principle ignoring those viewed as "parameters". However, action of VTS may increase the degree of other variables within atoms of the formulae. One need only inspect the suggested guards or formula schemes for substitution of a quadratic root to see that one may square various coefficients in x to obtain the total result of virtual substitution. For example, substitution of the first real root of the quadratic polynomial $f = ax^2 + bx + c$ with real type 1 into a relation $g = 0$ where g is the linear $a^*x + b^*$ involves usage of the formula scheme $2aa^*b^* - a^{*^2}b \geq 0 \wedge ab^{*^2} + a^{*^2}c - a^*bb^* = 0$. Of course $a^*, b^*, a, b, c \in \mathbb{Z}[\mathbf{u}]$, i.e. other polynomials in the parameters unrelated to x. Meanwhile the appropriate guard for this substitution is $a > 0 \wedge b^2 - 4ac > 0$, which similarly doubles the degree of b. This results in potential degree bloat for the other variables, which may be quantified or free. In the former case, this may frustrate the ability to use VTS further to eliminate the further quantified variables. In the latter case, this may only cause an issue where other algorithms are involved.

2.2 Universal Quantifiers

VTS is by presentation a tool to eliminate existential quantifiers, via aiming to find valid "examples" to satisfy the input formula in order to reconcile with the existential premise. Luckily, we have the equivalence

$$\forall x\ \Phi \equiv \neg\exists x\ \neg\Phi \tag{2.2}$$

to view a universally quantified formula in terms of an existentially quantified formula. Elimination of an existential quantifier produces a disjunction of the results of virtual substitution amongst the generated test points in x from that formula, so using (2.1) and (2.2) we obtain for a universally quantified formula:

$$\begin{aligned}
\forall x\ \Phi \equiv \neg\exists x\ \neg\Phi = \neg\left(\bigvee_{i=1}^{k} G(t_i) \wedge (\neg\Phi)[x \,//\, t_i]\right) \\
= \bigwedge_{i=1}^{k} \neg\left(G(t_i) \wedge (\neg\Phi)[x \,//\, t_i]\right) \\
= \bigwedge_{i=1}^{k} \left(\neg G(t_i) \vee \neg((\neg\Phi)[x \,//\, t_i])\right)
\end{aligned} \tag{2.3}$$

The second operand of the inner disjunction in that last expression can be read as "negation of the virtual substitution of t_i for x into the negation of the formula". The

following lemma demonstrates that, at least programmatically, we cannot discard $\neg\Phi$, because $\neg\Phi$ may generate a different structural test point set than Φ.

Lemma 8. *For an unquantified Tarski formula F over $\mathbb{Z}[x_1, \ldots, x_n]$, the structural test point set found by processing F by atposl and PC-to-TPs from [43] is not equal that from $\neg F$.*

Proof. Take $F \coloneqq (x_1 x_2 = 0)$. Then over x_2, F gives the structural test point set $\{[x_2 = -\infty], [x_2 = 0]\}$ via usage of atposl and PC-to-TPs. However, $\neg F = x_1 x_2 \neq 0$ gives $\{[x_2 = -\infty]\}$, due to $x_2 = 0$ being judged as an "inclusion point" by the former, and an "exclusion point" by the latter.

While the implementation of VTS in `QuantifierElimination` uses the distributivity of negation from (2.3) liberally in elimination of universal quantifiers, one will later observe this induces a lot of nested negation in the intermediate formulae to construct when there are several successive universal quantifiers to eliminate (one can observe this in Figure 2-3). There are some contexts in which viewing usage of VTS in this way is more academically canonical, such as production of witnesses (Section 2.5), however for the most part these continued negations are unnecessary overhead, and so refactoring of VTS in `QuantifierElimination` such that we merely process $\neg\exists \mathbf{x}\ \neg\Phi$ for a block of universal quantifiers $\forall \mathbf{x}$ is noted amongst elements to refactor in code in further work (Section 8.3). Nonetheless, the rest of this work discusses the coexisting existential and universal cases.

2.3 Blocks of Quantifiers & the VTS Tree

Due to multivariate VTS revolving around successive substitutions of structural test points, we easily inherit a canonical tree structure from substitutions of test points within any one block of quantifiers. The branches of this tree are structural test points, canonical tree levels correspond to one quantified variable each, and nodes are formulae owing to successive results of virtual substitution. The VTS tree is introduced by Košta in Section 6.2 of [43]. Definition 9 allows us to classify nodes of the VTS tree as objects with properties, and in this work we give more motive for an object oriented approach to the VTS tree.

Definition 9 (`IQER`). *An `IQER` (Intermediate Quantifier Elimination Result) is a node of the VTS tree. If it is the root node, it implicitly corresponds to the quantified problem for the current block of quantifiers. Otherwise, a node is the result of the virtual substitution of one structural VTS test point on another `IQER`. The Tarski formula associated with such a node is the result of that virtual substitution on the formula from the node above. In terms of an object-oriented implementation, we can associate the following properties to an `IQER`:*

- testpoint : *the associated test point used in virtual substitution to receive it (essentially the tree edge preceding the node), represented as a Maple list,*

- formulaSimplified : *its associated quantifier free (QF) Tarski formula, weakly simplified, which for the root node is the unquantified part of the input formula for QE (Φ) for this block of quantifiers, else the result of virtual substitution of its structural test point* testpoint *on the formula of the parent* `IQER` *above,*
- structuralSubstitution : *the associated quantifier free Tarski formula stored in structural form,*
- guardFormula : *a Tarski formula representing the guard for the substitution of* testpoint *to receive this* `IQER` *— only stored where structural form is relevant for future incrementality,*
- parent : *its parent* `IQER`*,*
- children : *an* `Array` *of its children* `IQER`*s, where populated,*
- level : *a non negative integer representing its level (viewed as a node of the VTS tree),*
- futureTestpoints : *a (mutable) set of structural test points for future elimination on this node in the next variable, which may be undefined until computed,*
- cad_formula : *an Extended Tarski Formula that overrides this* `IQER`*'s formula where CAD is used to deduce its equivalent (Chapter 4).*

Algorithm 1 represents the constructor method for an `IQER`. It takes a variable number of arguments in order to accommodate construction of the root `IQER`, which holds fewer properties, hence requiring fewer (and differing) arguments. Construction of an `IQER` of a positive level performs the virtual substitution of the best available test point from the `IQER` above into the formula held by that `IQER`, via the method getNextTestpoint implementing the strategy suggested by Section 2.3.3. This virtual substitution takes into account the quantifier for the block of variables for VTS to eliminate such that it uses the negations implied by (2.3). Hence we realise the construction of a node of the VTS tree in terms of the assignment of the majority of its properties at the point of construction.

Via (2.1) and (2.3), we have that elimination of existential quantifiers is canonically linked with forming a disjunction of the results of virtual substitution, while elimination of universal quantifiers is canonically linked with forming a conjunction of the results of (negation of) virtual substitution. The distributivity of quantifiers is such that

$$\exists x\ A \vee B \Rightarrow (\exists x\ A) \vee (\exists x\ B) \tag{2.4}$$

and similarly

$$\forall x\ A \wedge B \Rightarrow (\forall x\ A) \wedge (\forall x\ B), \tag{2.5}$$

but

$$\exists x\ A \wedge B \nLeftarrow (\exists x\ A) \wedge (\exists x\ B). \tag{2.6}$$

Algorithm 1 Constructor for an `IQER`

Input: Φ a Tarski formula, **OR** Q the quantifier symbol for the current block of quantifiers, vars the `Array` of corresponding quantified variables, P the parent `IQER`

Output: I, a new `IQER`, child of P

1: **procedure** `IQER`(Q, vars, P)
2: **if** 1 arguments was passed **then** ▷ Defining the root `IQER`
3: $I \mapsto \text{level} \leftarrow 0$
4: $I \mapsto \text{formulaSimplified} \leftarrow \Phi$
5: $I \mapsto \text{guardFormula} \leftarrow \text{NULL}$
6: **else**
7: $I \mapsto \text{level} \leftarrow P \mapsto \text{level} + 1$
8: $I \mapsto \text{parent} \leftarrow P$
9: $I \mapsto \text{testpoint} \leftarrow \text{getNextTestpoint}(\ P\ )$ ▷ Section 2.3.3
10: Let $T = I \mapsto \text{testpoint}$
11: $x \leftarrow \text{vars}[-I \mapsto \text{level} - 1]$ ▷ $x = x_{n-I\mapsto\text{level}}$
12: **if** Structural form requested **then**
13: **if** $p \mapsto \text{level} = 0$ **then**
14: $\Psi \leftarrow p \mapsto \text{formulaSimplified}$
15: **else**
16: $\Psi \leftarrow p \mapsto \text{structuralSubstitution}$
17: **end if**
18: **if** $Q = \exists$ **then**
19: $I \mapsto \text{structuralSubstitution} \leftarrow \Psi[x \,//\, T]$, with simplification under structural form
20: $I \mapsto \text{guardFormula} \leftarrow \text{guard}(\ T\ )$
21: $I \mapsto \text{formulaSimplified} \leftarrow \text{simplify}($ $I \mapsto \text{guardFormula} \wedge I \mapsto \text{structuralSubstitution}\ )$
22: **else**
23: $\Psi \leftarrow \neg\Psi$
24: $I \mapsto \text{structuralSubstitution} \leftarrow \neg(\Psi[x \,//\, T])$, with simplification under structural form
25: $I \mapsto \text{guardFormula} \leftarrow \text{guard}(\ T\ )$
26: $I \mapsto \text{formulaSimplified} \leftarrow \text{simplify}($ $I \mapsto \text{guardFormula} \vee I \mapsto \text{structuralSubstitution}\ )$
27: **end if**
28: **else**
29: **if** $Q = \exists$ **then**
30: $\Psi \leftarrow p \mapsto \text{formulaSimplified}$
31: $I \mapsto \text{formulaSimplified} \leftarrow \text{simplify}(\ \text{guard}(\ T\ ) \wedge \Psi[x \,//\, T]\ )$
32: **else** ▷ $Q = \forall$
33: $\Psi \leftarrow \neg(\ p \mapsto \text{formulaSimplified}\ )$
34: $I \mapsto \text{formulaSimplified} \leftarrow \neg(\ \text{simplify}(\ \text{guard}(\ T\ ) \wedge \Psi[x \,//\, T]\ )\ )$
35: **end if**

Algorithm 1 Constructor for an `IQER`, Part 2

```
        end if
        Add I as a child of P
        return I
    end if
end procedure
```

and similarly

$$\forall x\ A \vee B \Leftrightarrows (\forall x\ A) \vee (\forall x\ B) \tag{2.7}$$

Via assuming the input formula Φ is in prenex form, these are the only facts we need to use, i.e. we do not need similar facts about other boolean operators. These facts about distributivity imply VTS forming a tree structure on `IQER`s only within any one *block* of quantifiers. Via continued usage of (2.1) with (2.4) on Φ, we have

$$\exists x_{n-m+1} \dots \exists x_n\ \Phi(x_1, \dots, x_n) \equiv \exists x_{n-m+1} \dots \exists x_{n-1} \bigvee_{i=1}^{k} G(t_i) \wedge \Phi[x_n /\!/ t_i]$$

$$\equiv \exists x_{n-m+1} \dots \exists x_{n-2} \bigvee_{i=1}^{k} \exists x_n\ G(t_i) \wedge \Phi[x_n /\!/ t_i] \tag{2.8}$$

and similarly (2.3) with (2.5) imply

$$\forall x_{n-m+1} \dots \forall x_n\ \Phi(x_1, \dots, x_n) \equiv \forall x_{n-m+1} \dots \forall x_{n-1} \bigwedge_{i=1}^{k} \neg G(t_i) \vee (\neg\Phi)[x_n /\!/ t_i]$$

$$\equiv \forall x_{n-m+1} \dots \forall x_{n-2} \bigwedge_{i=1}^{k} \forall x_n\ \neg G(t_i) \vee (\neg\Phi)[x_n /\!/ t_i] \tag{2.9}$$

but via the latter facts about distributivity that do not work ((2.6) and (2.7)), we have

$$\forall x_{n-m+1} \dots \forall x_{n-1} \exists x_n\ \Phi(x_1, \dots, x_n) \equiv \forall x_{n-m+1} \dots \forall x_{n-1} \bigwedge_{i=1}^{k} G(t_i) \wedge \Phi[x_n /\!/ t_i]$$

$$\not\equiv \forall x_{n-m+1} \dots \forall x_{n-2} \bigwedge_{i=1}^{k} \forall x_{n-1}\ G(t_i) \wedge \Phi[x_n /\!/ t_i],$$

hence why we can only form a tree structure within blocks, and the quantifier alternation as above impedes one "global" tree structure. The above equivalences mean we form a constantly expanding disjunction or conjunction of the results of virtual substitution which correspond to the *current* leaves of the VTS tree for elimination of any one block of quantifiers. The inner operand $\exists x_n\ G(t_i) \wedge \Phi[x_n /\!/ t_i]$ from (2.8) expands to

$$\bigvee_{j=1}^{s} G(\tau_j) \wedge (G(t_i) \wedge \Phi[x_n /\!/ t_i])[x_{n-1} /\!/ \tau_j]$$

for another elimination set $\{\tau_1, \ldots, \tau_s\}$, $s > 0$, in x_{n-1} for $G(t_i) \wedge \Phi[x_n \,//\, t_i]$ using (2.1) such that the VTS tree can be visualised as Figure 2-2 for the existential case. Figure 2-3 visualises the similar universal case. Despite expanding e.g. a disjunction within a disjunction, `QuantifierElimination` is always cognizant to not create a needlessly nested formula, i.e. any produced disjunction for output should be as flat as possible. Note that while the variable ordering used to obtain `IQER`s of identical levels here must be the same, this is for simplicity of implementation (with a view to the poly-algorithm discussed later) rather than of theoretical significance — the fact that differing traversals of the VTS tree here result in different constraints and benefits on variable ordering is discussed later in Section 2.3.2.

Because the cases for each type of quantifier in VTS are fundamentally different, we need only consider the case for one block of quantifiers without loss of generality when discussing VTS. The recursive nature of VTS in that quantifier elimination directly returns a Tarski formula means that the output Tarski formula from VTS on one block of quantifiers replaces the innermost part of the formula up to and including that block of quantifiers, and one continues with the next block. Further, whenever VTS is concerned (in particular this chapter and Chapter 4), we always consider the case for one type of quantifier symbol $Q \in \{\exists, \forall\}$, i.e. assume the quantified input Tarski formula is the prenex $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$, with $m > 1$ similarly quantified variables amongst $n \geq m$ total variables.

Definition 10 (Meaningful Truth Value). *A meaningful truth value for a block of quantifiers $Qx_{n-m+1} \ldots Qx_n$ in VTS is* true*, if $Q = \exists$, or* false *if $Q = \forall$.*

Definition 11 (Leaf `IQER`s). *An `IQER` is a* genuine leaf *of the VTS tree for a block of quantifiers $Qx_{n-m+1} \ldots Qx_n$ if its associated quantifier free formula is* true *or* false*, or if it has level $j - i + 1$. It is a* meaningful leaf *if it is an `IQER` of positive level such that:*

- *holding the quantifier free formula* true *when $Q = \exists$,*
- *or holding the quantifier free formula* false *when $Q = \forall$.*

Or in other words it is a meaningful leaf if it holds a meaningful truth value for the quantifier of the current block of variables.

Due to the distributivity of quantifiers into boolean formulae, it will further be completely be appropriate to view any one `IQER` as an implicitly quantified formula. In particular, it is an individual QE problem in itself quantified by the current block of quantifiers, the number of which is lessened by its level in the tree. As such, it is not implicitly quantified with anything from the innermost block of quantifiers if it is a leaf (Definition 11). In the context of a current innermost block of quantifiers $Q_i x_i \ldots Q_j x_j$, where $n - m + 1 \leq i \leq j \leq n$, $Q_i = Q_k = Q_j$, $\forall k = i, \ldots, j$ an `IQER` can be seen to represent the QE problem

$$Q_i x_i \ldots Q_{j-I\mapsto\text{level}} x_{j-I\mapsto\text{level}}\ I \mapsto \text{formulaSimplified} \tag{2.10}$$

via distributivity of that block of quantifiers into the disjunction or conjunction formed by VTS. Obviously $I \mapsto \text{level} \leq j - i + 1$, and a genuine leaf `IQER` is implicitly quantified

by nothing from the innermost block of quantifiers. Due to this characterisation, each operand of the form $G(t_i) \wedge \Phi[x_n // t_i]$, $i = 1 \ldots, k$ of (2.8) is an `IQER`, and $\neg G(t_i) \vee (\neg\Phi)[x_n // t_i]$, $i = 1 \ldots, k$ of (2.9) is similarly an `IQER`.

Definition 12 (Ineligible `IQER`). *An `IQER` is* ineligible *if it is a leaf of the VTS tree such that, in the context of (2.10), the maximum degree of any factor of all atoms of* $I \mapsto$ formulaSimplified *in* x_k *exceeds* $0 \leq m \leq 2$, $\forall k = i, \ldots, j$, *where* m *is the value of the keyword option `'MaxVSDegree'` passed to a top level QE function in `QuantifierElimination` using VTS. (By default `'MaxVSDegree'` = 2.)*

Definition 12 is essentially the analogous notion of a "degree violation" in other work on VTS. In particular it is a degree violation with respect to the QE problem defined by a particular `IQER` in light of any imposed variable ordering. `'MaxVSDegree'` controls the maximum degree VTS will use to generate structural substitution sets for `IQER`s from irreducible polynomials after factoring. That is, if `'MaxVSDegree'` = 1, then VTS will attempt to factor polynomials from within relations in `IQER`s to irreducible factors of 1, and if it finds any irreducible factors of degree 2, then the `IQER` is identified as ineligible. `IQER` selection strategy (Section 2.3.1) is actually the way in which `IQER`s are judged as ineligible — the `IQER` selection strategy function will return -1 if it iterates across all `IQER`s in the container iqers and finds them all to be ineligible in light of the value of `'MaxVSDegree'`. `'MaxVSDegree'` = 0 trivially sends QE to perform CAD on all `IQER`s available, and if this is at the start of any QE, then QuantifierEliminate becomes equivalent to usage of PartialCylindricalAlgebraicDecompose.

Definition 13 (Canonical Boolean Operator). *The* canonical boolean operator *for usage of VTS in elimination of a block of quantifiers* $Qx_{n-m+1} \ldots, Qx_n$ *is "*or*"* $(\bigvee)$ *if* $Q = \exists$, *or "*and*"* $(\bigwedge)$ *if* $Q = \forall$.

If we continue VTS to the point where we create at least one meaningful leaf, we are certainly done, and the quantifier free equivalent of the whole quantified formula $Q_{n-m+1}x_{n-m+1} \ldots Q_n x_n\ \Phi$ is the meaningful truth value of that leaf. Otherwise, the equivalent of the innermost block of quantifiers on the input formula is the disjunction/conjunction of all genuine leaves of the tree, i.e. those of level $j - i + 1$, and if there are any ineligible `IQER`s, then VTS cannot eliminate all quantifiers from this block alone. Chapter 4 discusses the ways in which `QuantifierElimination` moves to use CAD from VTS in the presence of ineligible `IQER`s, which is dependent on whether VTS is in the last block of quantifiers or not.

If we are able to deduce the quantifier free equivalent of the innermost block of quantifiers, this quantifier free equivalent replaces the input formula as the next input for VTS with the next block of quantifiers. In this case, we form a layered "Christmas" tree (Figure 2-1) out of usage of VTS on multiple blocks of quantifiers, however there is no direct correspondence between nodes of distinct trees beyond the fact that one tree follows from another. The root of one tree is not a particular node from the tree above, but in some sense the collection of all leaves of the tree above. When there is only one block of quantifiers to eliminate, VTS forms just one tree, and one of the reasons behind identifying the tree structure behind VTS is to enable the methodology of Section 2.5.

2.3.1 IQER Selection Strategy

The available IQER selection strategies in QuantifierElimination are controlled by the keyword option 'mode', which takes exactly one of the symbols 'depth' or 'breadth'. This corresponds to traversing the VTS tree depth-wise or breadth-wise respectively, hence selecting the IQER of maximum or minimum depth respectively. There are two further static tiebreakers appended to this metric:

- IQERs that already have generated test points are prioritised,
- IQERs such that their formula is of a lower maximum degree in the next quantified variable to eliminate $x_{n-I\mapsto\text{level}}$ are prioritised.

The last tiebreaker is greedy in the sense it minimises the maximum possible asserted degree amongst the test points to generate, hence minimising the potential degree bloat in the associated virtual substitutions. This reconciles with the aim to propagate VTS as far as possible.

VTSIQERSelectionStrategy is the function that returns the index of the next suggested IQER to propagate VTS on with respect to the metric suggested by the above. It iterates over the whole "iqers" container, storing the index of the best IQER together with the data about the best possible level & minimal degree found in order to enable comparison. The "iqers" container is essentially the equivalent of a container of "work" nodes in [43]. The iqers container is programmatically a QEContainer, which is a rudimentary object supporting typical storage object operations such as

- addition (push),
- arbitrary removal of elements at a position (pop),
- iteration,
- peeking (iqers[i]), etc.

As an object, under the hood it uses an Array for its storage, where an Array does not naturally support removal, hence the modification. Note that IQERs with an empty set of future test points futureTestpoints are ejected from this container before strategy is called on it, i.e. if we use the last test point from an IQER in propagation of VTS, it is removed from the container at this point. Hence IQERs attributing further test points should remain within the container, enabling the second tiebreaker.

Usage of this container could in practice be replaced with a heap, where addition of an IQER is $\mathcal{O}(\log k)$ as opposed to $\mathcal{O}(1)$, but selection and/or extraction of the "maximal" IQER is $\mathcal{O}(1)$ instead of $\mathcal{O}(k)$. If it were to be replaced by a heap, we must be cognizant that an IQER is not immediately amenable to further propagation of VTS if the formula it holds is of excessive degree. VTSIQERSelectionStrategy returns -1 if the container "iqers" is non empty but contains only IQERs of excessive degree (ineligible IQERs, in light of the value of 'MaxVSDegree'), as to notify quantifier elimination to switch to CAD in some capacity. Therefore if usage of a QEContainer was to switch to a heap, it really needs to be two heaps, one holding IQERs amenable to further

VTS, and the other ineligible `IQER`s. The latter heap's sorting function could be one reconciling with VTS to CAD strategy when the poly-algorithm is concerned (Section 4.2).

A further mitigation is that once a maximal `IQER` is selected, `QuantifierElimination` allows for propagation of VTS by one test point at a time from the generated set of test points associated to that `IQER`. Therefore the `IQER` shouldn't be extracted immediately from the heap as long as it attributes test points. When an `IQER` is selected, we generate the set of structural test points on that `IQER` if it didn't already exist, which in principle modifies data relating to the tiebreakers, however this generation cannot ruin the "heap condition". However, in the current implementation of `QuantifierElimination`, variable strategy is dependent on `IQER` strategy, which means that the last tiebreaker is not highly agreeable with a heap, which may require sifting amongst subheaps when variables are reordered. This tiebreaker is not highly compatible with a heap, which would require resorting of subheaps when variables are reordered. However, removing this tiebreaker makes it difficult to realise a truly greedy codependent variable & `IQER` strategy such that one ensures in best possible terms that one can traverse a path to a (meaningful) leaf `IQER` as quickly as possible without impediments in terms of degree. This codependency is discussed further in Section 2.3.2 and formalised by Code Fragment 4.

Because selection strategy on a `QEContainer` involves iteration over the container, strategy is immediately mutable, but considering evolutionary operations (Algorithms 47, 49) always regenerate the container of work `IQER`s, i.e. there is no reason to retain an existing one, and so this fact is not particularly to the `QEContainer`'s credit.

Unlike the case for CAD cell selection strategy, which is discussed in Section 3.9, there is no reason for a *strict* subset of `IQER`s to be removed from the VTS tree (and hence discarded from the container "iqers"), because propagation of truth values in the same sense is trivial. A VTS tree only corresponds to one block of quantifiers, and hence there is only ever one meaningful truth value to ever propagate within one block. If we receive a meaningful truth value, this truth value would propagate all the way to the root, hence we could discard all `IQER`s from the tree, because we are ready to terminate quantifier elimination, having found the quantifier free equivalent. Therefore arbitrary removal of non maximal elements from whatever data structure is used to store "work" `IQER`s is not a relevant operation, in the same way as it is for CAD (Section 3.9).

In total, replacement of usage of a `QEContainer` with a heap could be a win due to pairs of addition & selection operations reducing to $\mathcal{O}(\log k)$ from $\mathcal{O}(k)$, with one caveat being that with a heap, non trivial work is always expended to add elements to the structure, which may have gone to waste if termination criteria is met (i.e. we receive a meaningful truth value, and no further usage of "work" `IQER`s is required). A further caveat is that strategy could no longer be dependent on anything highly mutable, such as the current variable ordering.

2.3.2 VTS Variable Strategy

VTS in `QuantifierElimination` implements a greedy variable strategy based on the next selected `IQER` for propagation of VTS, to reconcile with the usual aims of `QuantifierElimination` to implement greedy strategies. Via usage of VTSIQERSelectionStrategy from the above section, we know that VTSIQERSelectionStrategy selects a non ineligible `IQER` I for elimination, such that the maximum degree of any factors of any polynomial in I is at most the value of the keyword option `'MaxVSDegree'`. Let $0 \leq j \leq m$ is the maximum level of any `IQER` when the function implementing VTS variable strategy, VTSVariableStrategy is called. VTSVariableStrategy insertion sorts the variables $x_{n-m+1}, \ldots, x_{n-j}$ in terms of the maximal degree of each variable in I, modulo the fact that only quantifiers with the same quantifier symbol commute. More specifically, VTSVariableStrategy attempts to minimise the maximal degree of the variable to next use for test point generation (and hence elimination) on I, except that it does not attempt to sort variables not contained within I towards the back of the variable ordering, i.e. it does not aim to minimise the maximal degree to eliminate next all the way to 0. This would result in generation of trivial test point sets that go nowhere to actually performing non trivial quantifier elimination on I. The criteria on j ensures that we do not sort variables that are "fixed", which would ruin the integrity of the VTS tree. The action of VTS implies that elimination of a variable x_i may double the degrees of the variables $x_1, \ldots, x_{i-1}$ when quadratic elimination is concerned, however this preserves their sorting via action of virtual substitution on all atoms of a formula.

Insertion sort is appropriate because the number of variables is assumed to be few, but further scope for variable strategy is restricted by commutativity only of quantifiers with the same quantifier symbol (which is all there is to do in VTS due to acting within any one block of quantifiers), which mitigates the headline $\mathcal{O}(n^2)$ complexity of insertion sorting further, but the main grounds for usage of insertion sort is that it is convenient when acting upon the `Array` of variables in-place, which is the usual semantics for action of variable strategy on an `Array` of variables within `QuantifierElimination`, also eschewing creation of intermediate data structures for garbage collection.

Usage of this greedy variable strategy is intrinsically designed to compliment and reinforce usage of depth-wise traversal of the VTS tree in propagation of VTS, because it aims to propagate VTS as far as possible without usage of CAD, because minimizing the degree of the elimination sets minimizes the degree bloat on the formula of the produced `IQER`. Usage of this variable strategy in conjunction with depth-wise traversal means we certainly attempt to branch all the way to a leaf `IQER` as soon as possible (hopefully a meaningful leaf). The path to this leaf entirely defines and fixes the variable ordering for usage of VTS to create the rest of the tree, so one disadvantage of usage of this strategy is that it may not accommodate `IQER`s created elsewhere in the tree well past the first traversal towards a leaf, but usage of depth-wise traversal in poly-algorithmic QE on the VTS tree for a last block of variables may benefit from the set of ineligible `IQER`s being of disjoint level, due to the aim of having high level `IQER`s to process by CAD first to minimize the number of variables in CAD at any one time. The codependency of variable strategy and `IQER` selection strategy means

that one must call variable strategy via the root `IQER` before attempting propagation of VTS (Code Fragment 4), which will sort variables in terms of the initial input degrees of polynomials. From there, the action of VTS in terms of the chosen `IQER`s will largely mean the variables stay sorted with respect to the path traversed by successive `IQER`s in terms of depth-wise traversal of the VTS tree, due to the degree bloat of virtual substitution acting the same on all further variables. VTSVariableStrategy in `QuantifierElimination` examines degrees of polynomials from an `IQER` *without* factorisation, to minimise the amount of factorisation done as a result of strategy, and because being able to begin VTS from input that usually features no factorisation largely ensures that VTS continues to identify eligible `IQER`s with the action of VTS' degree bloat acting transitively.

2.3.3 Test Point Selection Strategy

QuantifierEliminate associates future test point sets to `IQER`s, via the futureTestpoints property. Once the set of structural test points has been generated for an `IQER`, this set remains, with individual test points being removed from the set once used to form a child `IQER`. Because propagation of VTS in `QuantifierElimination` can be done via individual test points in order to form one child `IQER` at a time, test point selection on a selected `IQER` is tangible and relevant. The suggested & current metric for selection of test points in VTS is via the following tiebreakers:

- If a test point is substitution of $\pm\infty$, i.e. the test point is $[0, x, \pm\infty]$ for some x, choose this,
- else if the test point has no infinitesimal (i.e. ε) and the asserted degree for virtual substitution is lowest seen thus far, choose this,
- else choose the test point with lowest possible asserted degree for the test point (amongst those test points with ε).

Usage of this strategy achieves the goal of being a "greedy" test point selection strategy, as we attempt to minimise the resulting degree of the formula resulting from substitution, else the number of atoms in the formula to create as a result of substitution.

- Substitution of test points featuring $\pm\infty$ via subs-inf-rel requires no usage of pseudoremainder, producing a formula purely predicated on signs of coefficients and reducta of polynomials from the formula to substitute into. There is also no associated degree bloat.
- The next preferable type of test point is one of low (asserted) degree without the infinitesimal ε. Test points featuring ε require epsilon expansion on formulae, increasing the number of atoms in the formula to eventually perform weak substitution on. Formulae from epsilon expansion via expand-eps-at have a maximum degree equal to that of the formula acted upon. Hence choosing a test point without ε is merely to minimise the number of atoms to substitute into.

- Choosing a test point of minimal asserted degree minimises the degree bloat out of the pseudoremainder and formula scheme combination to achieve the required weak substitution. For example, substitution of a polynomial of asserted degree 1 reduces entirely to pseudoremainder inducing no degree bloat. The formulae schemes for substitution of a quadratic test point into a linear polynomial induce less degree bloat than those from cubics.

The future test point set for an `IQER` is the direct result of usage of PC-to-TPs and atposl on the formula for that `IQER`. In Maple, it is a `MutableSet` to reconcile with the continuing need for mutable data structures. getNextTestpoint is the method implementing the described test point selection strategy on an `IQER` — it also generates the set of test points for the `IQER` via PC-to-TPs and atposl on its simplified formula. In particular this type of set supports deletion of expressions, i.e. one can remove a test point once selected for substitution. There is no reason for removal of arbitrary non maximal test points, so conversion of these sets to a heap of test points is entirely plausible. As per the usual observation on usage of heaps, not all test points may be used per `IQER`, so the work expended in addition of test points using the ordering implied by the above (as usual, trivial to evaluate) tiebreakers may be lost on early termination. But as per usual one may expect the lesser complexity of construction of any one heap beats the continuing $\mathcal{O}(k)$ expense of selection amongst k test points for an `IQER` on average.

2.4 Tarski Formulae for VTS

Tarski formulae are intrinsic and vital to the operation of VTS. In particular, we must be able to build such formulae efficiently, because every `IQER` stores a formula that is the result of virtual substitution of a test point into the formula from an `IQER` above. Additionally, we may wish to simplify the implicit quantifier free formula representing sufficiency of the VTS tree in order to deduce that we can terminate the algorithm (as early as possible). A discussion of simplification can be found in the following Section 2.4.2.

Maple offers the `Array` as a mutable data structure for storage of expressions. We use this structure for storage of Tarski formulae instead of formulae via the inert `And` and `Or` symbols in Maple as the former's mutability is essential to allow for efficient building & modification of formulae, at the very least to accommodate this package's version of incrementality (Section 5.1.2), which may require insertion of formulae into an existing formula (at any `IQER`). Such modification of a Tarski formula using inert symbols actually requires rebuilding an entirely new expression, whereas usage of an `Array` does not. On the other hand, simplifying formulae may require us building a formula "incrementally", because we do not know ahead of time how many operands the formula will have, and here usage of `Array`s is obvious. In total, usage of `Array`s allows us to avoid incrementally building formulae in a series of immutable data structures which would incur a $\mathcal{O}(k^2)$ cost (for k the total number of atoms in the expression to build) due to garbage collection of discarded intermediate data structures. Instead

we can achieve $\mathcal{O}(k)$ with usage of the mutable `Array` in avoidance of such garbage collections.

Hence, a `TFArray` is the Maple *type* that refers to a quantifier free prenex Tarski formula stored in `Array` format. This is either an atom (Definition 1), or an `Array` with `And` or `Or` as the first element, with at least two `TFArray`s as the successive elements. A `TFInert` is the Maple type referring to any quantifier free Tarski formula using inert `And` or `Or`, or other allowable inert Maple boolean operators including `Xor`, `Implies`, `Not` (with appropriate numbers of arguments respectively). A `TarskiFormula` type is similar to `TFInert`, but allows for quantifiers preceding subformulae via the inert `exists` or `forall`. Lastly, a `PrenexQuantifiedTF` is the equivalent type to (1.1). These types reconcile with formulae over the integers as per Definition 3.

The `TFArray` and `TFInert` types are the most relevant with respect to computation, as the former is used throughout QE "under the hood" in VTS, and the latter is used in output (which we only need to build once). Unnecessary nesting of formulae would be the case where a conjunction appears as a genuine argument to another conjunction, i.e. a `TFArray` with operator `And` appearing at the top level of another `TFArray` representing a conjunction. While this is not disallowed within the type for a `TFArray`, in the general case the builder function for `TFArray`s, buildTFArray, ensures that no unnecessary nesting occurs for the sake of efficiency. Unnecessary nesting introduced by the user via quantified input is also guaranteed to be done away with in conversion of the formula Φ to a `TFArray` — note that the `TFArray` type only admits prenex formulae, i.e. the only operators are `And` and `Or`, so "unnecessary" really refers to a matter of efficiency.

2.4.1 Delayed Evaluation of Virtual Substitution

One of the further suggestions of the work [43] is delayed evaluation of virtual substitutions into a non atomic formula. The idea at least owes to short circuiting of disjunctions and conjunctions. For example, if we have the disjunction

$$\bigvee(\Psi_1(x,\mathbf{u}),\ldots,\Psi_k(x,\mathbf{u}))[x\,//\,t] = \bigvee(\Psi_1(x,\mathbf{u})[x\,//\,t],\ldots,\Psi_k(x,\mathbf{u}))[x\,//\,t]) \quad (2.11)$$

then if an early $\Psi_j(x,\mathbf{u})[x\,//\,t] \equiv \mathit{true}$ for $1 \leq j < k$, then evaluation of the virtual substitutions $\Psi_i(x,\mathbf{u}))[x\,//\,t]$ for $j < i \leq k$ are superfluous, because we already know that the disjunction is equivalent to *true*. Virtual substitution generally reduces to taking pseudoremainders, giving impetus to avoidance of further evaluations.

Delayed Evaluation in Maple

As Computer Algebra software, Maple offers features such as unevaluation and inertisation. In general, Maple will perform full evaluation on all arguments to any procedure recursively, and from left to right (in terms of positional arguments). Therefore it is not sufficient to supply $\Psi_1(x,\mathbf{u})[x\,//\,t],\ldots,\Psi_k(x,\mathbf{u})[x\,//\,t])$ from (2.11) to a variadic function to build the disjunction of such, because even with the unevaluation feature of

Maple, such a procedure will evaluate all of these. One can however manually inertise these substitutions. The expression

$$\texttt{':-inertSubsWeak'}(\Psi_j, T)$$

in Maple is an example of such a manual inertisation, where it represents $\Psi_j(x, \mathbf{u})[x // t]$ if T is the structural test point representing $[x // t]$ and Ψ_j is atomic. If Ψ_j is not atomic then the appropriate inertisation involves `':-inertSubsWhole'`. The usage of `':-...'` is Maple syntax where:

- `:-` refers to the global instance of the symbol `inertSubsWeak` in that Maple session, in case it would be assigned to in source code for `QuantifierElimination` (or the source code for any module/package using it),
- the quotes `'...'` unevaluate the symbol `inertSubsWeak` such that it is not interpreted in terms of any assignment to `inertSubsWeak` by the user in a global sense.

These inertisations can also be defined to be typed as a `TFArray`, as would be required for these constructs to be passed to appropriate intermediate VTS functions using Tarski formulae. The expression `':-inertSubsWeak'`(Ψ_j, T) can be viewed as a wrapper for the actual call to evaluate, hence replacement of the "operator" `inertSubsWeak` with substituteWeakRelation evaluates the actual virtual substitution.

buildTFArray is the procedure that builds arbitrary formulae in `QuantifierElimination`, taking the name of an operator `And` or `Or`, and then being variadic, accepting a sequence of formulae. The main purposes of the function are to prevent nesting of formulae, weakly simplify atoms, but also performs the evaluation of inertised substitutions amongst the sequence of formulae, as it iterates across this sequence. Expressions such as lines 31 and 34 of Algorithm 1 are performed via usage of buildTFArray, such that $\Psi[x // T]$ is inertised upon passing to buildTFArray, and the virtual substitution of T into the (potentially non atomic) formula Ψ is only computed when guard(T) is non *false*.

Košta's "formulae schemes" [43] to describe substitution of a root in x from a polynomial f into a polynomial g such that $1 < \deg_x(g) < \deg_x(f) \leq 3$ can be recursive on virtual substitution. For example, the scheme

$$D_g \geq 0 \wedge (f = 0)[x // \beta_1] \vee (f = 0)[x // \beta_2]$$

describes the result of substitution of the only root of f into $g = 0$ where $\deg_x(f) = 3$, $\deg_x(g) = 2$, and f is of real type 1[1]. formulaScheme in `QuantifierElimination` is essentially a lookup table providing the formula to build corresponding to the enumeration of Košta's formulae schemes, and directly includes the inertisations such that the

[1]Additionally, $D_g = \text{discrim}_x(g)$, and β_i is the structural test point describing the ith root of g, $i = 1, 2$ where g is asserted to have real type 1.

return value for this scheme is

$$\text{buildTFArray}(\texttt{And},\ D_g \geq 0,\ \text{buildTFArray}(\texttt{Or},\ \texttt{`:-inertSubsWeak'}(f = 0, \beta_1),\ \texttt{`:-inertSubsWeak'}(f = 0, \beta_2))).$$

Formulae schemes are only recursive in this way with respect to cubic elimination, which in total is a future addition to `QuantifierElimination`.

A similar implementation of delayed substitution for the formulae to be held by `CADCell`s in PartialCylindricalAlgebraicDecompose is discussed in Section 3.4.2. This is briefly contrasted with the delayed substitution in VTS. Note that in practice this delayed evaluation of substitutions is not implemented when structural form is requested in order to return a `QEData` object for further incrementality, merely because to realise insertion and deletion with atomic position (Section 5.1.1), it is easiest if the formula at any one atomic position is a genuine formula rather than "inert". One may be able to extend the implementation by evaluating inertised substitutions where appropriate in structural formulae when relevant for atomic position.

2.4.2 Simplification of Tarski Formulae

Expressions are central to Computer Algebra, whether they be objects such as polynomials, rational functions, or transcendental functions where in each case there is scope for interpretation to the term "simplify". One need only take an example such as $\frac{x^{100}-1}{x-1}$ to notice that (ignoring the discussion on what happens at $x = 1$!), evaluating the actual division will result in $x^{99} + \cdots + 1$, which is dense with 100 terms. The two metrics "give me something without a fraction" and "give me something with less distinct operands" are conflicting and incompatible, so one really does need to give a concrete meaning to "simple" to be able to call an expression "candid".

Definition 14 (Candid). *A* candid *expression is one that is not equivalent to an expression that visibly manifests a simpler expression class. [60]*

We introduce the concept of a "simplifier". One notes the built in Maple function simplify, which acts upon algebraic expressions of a wide range of types.

Definition 15 (Simplifier). *A* simplifier*, S, is a function that maps objects in some set T to those in T, and for all $t \in T$ meets the following specification: for all t in T,*

$$S(t) \sim t,$$

and

$$S(t) \leq t,$$

where "$\leq$" owes to some metric of "simplicity".

Via this definition, an expression t is "candid" if $S(t) = t$. Forward, S refers to a simplifier on the set of Tarski formulae T with no further definition than as above.

With respect to Tarski formulae, some such metrics seem obvious, such as length of a formula — the total number of atoms within. It is tempting to say that a shorter

formula is always better. This certainly can be trivially true, such as for formulae such as $x = 0 \vee x < 0$, which should almost certainly be simplified to $x \leq 0$, considering such a relational operator is allowable in our system. In particular $x = 0 \vee x < 0$ is unpreferable for test point generation in VTS compared to $x \leq 0$, where the former would contribute four candidate test points and the latter two candidate test points via atposl, as a result of the distinction between an "isolated point, with strict lower and upper bounds" and a "weak upper bound". It is also somewhat obvious that the user would prefer to see this single atom in output if it came to it.

On the other hand, consider a formula such as $f = 0 \vee g = 0$. For the purposes of example, assume f, g are irreducible, and of non trivial degree. This formula *can* be simplified to $fg = 0$, which is of a lesser length, but of a higher degree (as in the associated left hand side is of a higher degree). Because Maple allows a product of polynomials to exist without expansion, storing the left hand side of the relation as such a product means we do not have to worry about introducing unnecessary expense to factorisation of the polynomial later (as both VTS and CAD alike generally require). Whether a user would prefer to see $fg = 0$ (with or without expansion of fg) or $f = 0 \vee g = 0$ upon output is instead subjective.

These are all points that need to be taken into account if one is to define "simple", and hence obtain an effective simplifier for Tarski formulae. On the other hand, we must consider the expense of simplification in terms of time. The atom $x^2 + 1 > 0$ is candidly equivalent to *true*. Ironically, to deduce as such, we can pose the QE problem $\forall x\ x^2 + 1 > 0$. More generally, the atom $f(\mathbf{x}) > 0$ is candidly equivalent to *true* as an atom if $\forall \mathbf{x}\ f(\mathbf{x}) > 0$, or candidly equivalent to *false* if $\neg\exists \mathbf{x}\ f(\mathbf{x}) > 0$. In non atomic terms, we have e.g. $xy = 0 \vee x^2 = 0 \equiv xy = 0$. Deducing the equivalence can be done by noting that $\gcd(xy, x^2) = x$, so the second operand is superfluous, but one can see that in general some non trivial operation is required in order to make such a deduction.

Definition 16 (Weak Simplifier). *A* weak simplifier *for a Tarski formula is one that maps atomic expressions free of variables to* true *or* false*, such as* $0 < 0 \mapsto$ false*, and* $1 = 1 \mapsto$ true*, and otherwise applies "short circuiting" rules to simplification of expressions below conjunctions and disjunctions, i.e.* false *is discarded as an operand below a disjunction, or* true *as an operand to a disjunction evaluates the simplification of the entire disjunction to* true*. Additionally, we prevent duplicate operands within formulae.*

Loosely, a "strong simplifier" should then be a simplifier that brings formulae closer in line to "candid" than a weak simplifier. `QuantifierElimination` only implements weak simplification for formulae, which especially has ramifications on VTS. Considering the presence of existing work in this area, the investigation of a stronger simplifier for `QuantifierElimination` is noted as important canonical further work for `QuantifierElimination` (Section 8.3), and poor simplification is noted as a particular reason why the performance of VTS in `QuantifierElimination` falls short of implementations similarly implementing VTS such as `SyNRAC` (Section 7.4.4).

One notes the Maple inbuilt functions is and coulditbe which essentially pose

the fully universal and fully existentially quantified questions on a relation as above. `QuantifierElimination` currently does not choose to use even these functions thus far, which would provide stronger simplification on *atoms*, but not necessarily across entire formulae. One notes that beyond a quantified problem, a question such as $\neg\exists \mathbf{x}\, f(\mathbf{x}) > 0$ implicitly features an SMT problem on *one atom*, of course over the theory of real arithmetic, so the problem is less general than that of QE, so in some sense we may not require full "recursion" on QE. On the other hand, in VTS, similar "SMT-like" questions asked about possible signs of polynomials (at-cs, at-cs-fac), such as the content of a polynomial, or a leading coefficient are posed, and `QuantifierElimination` chooses to solve these "SMT-like" queries via recursion on QuantifierEliminate. While SMT over the theory of real arithmetic is within the scope of `QuantifierElimination`, it would be best to consolidate the approach here towards an SMT solver for the theory of real arithmetic to solve such queries (again, specific algorithms for SMT over the theory of real arithmetic are noted as further work in Section 8.3). Even better, Definition 18 most succinctly summarises the desire in terms of simplification.

Definition 17 (Monte Carlo Algorithm). *A* Monte Carlo algorithm *is one that is "always fast, probably correct".*

Open Problem 18. *Can we replace "correct" in the above with "candid" to obtain a strong simplifier that we can deploy that attempts to expend no more work in simplification than the projected saving by performing such simplification?*

Open Problem 18 highlights that the work expended in simplification should be offset by the expected gains in actually simplifying such a formula. This is a sentiment identified multiple times in this work. In particular, strong simplification on the formula of one fully quantified ineligible `IQER` could be seen as somewhat needless if the strong simplification were to include involved usage of CAD, because CAD may be used to provide the candid quantifier free equivalent of that `IQER` later on anyway (e.g. via the poly-algorithm).

We move on to discuss the typical notions of "canonical" and "normal" in Computer Algebra.

Definition 19 (Canonical). *A representation is said to be* canonical *if every object has only one representation.*

Definition 20 (Normal). *A representation is said to be* normal *if the only representation of the object 0 is* `0`*.*

Normality becomes relevant when evaluating expressions featuring (real) algebraic numbers (Section 3.4.2), but these do not appear in the Tarski formulae for VTS. Definition 20 could be adapted and interpreted in the context of Tarski formulae by requesting that the only representation of an atom equivalent to *true* or *false* are *true* or *false* themselves. This is related to candid representations, but is less restrictive, because it only refers to atomic formulae.

Definition 21 (Locally Canonical). *A representation is said to be locally canonical (with respect to a certain context) if every object whose introduction does not change the context has only one representation.*

`QuantifierElimination` globally asserts that all relations are in a canonical form, such that they are always stored as $f\,\rho\,0$ where f is monic (if $\rho \in \{=, \neq\}$, otherwise f merely need have trivial integer content). VTS has no motive for storage of mere polynomials outside of relations, but CAD does, so CAD attempts to store canonical polynomials in bases alongside canonical relations in its notion of Tarski formulae. However, some frustration for a canonical form for polynomials in CAD arises due to the representation of real algebraic number coefficients.

Tarski Formulae in Structural Form

As operands of any non atomic Tarski formula commute, $f_1\,\rho\,0 \ltimes f_2\,\rho\,0$ and $f_2\,\rho\,0 \ltimes f_1\,\rho\,0$, $\ltimes \in \{\wedge, \vee\}$ are functionally equivalent. Luckily, there is never any reason to commute such operands of a formula even after applying the distributivity of virtual substitution as per the right hand side of (2.11), hence the ordering of operands of a formula is preserved under virtual substitution. The very general incrementality in VTS (and CAD) of `QuantifierElimination` works via the concept of *structural form* for Tarski formulae (Section 5.1.1). To acquire a formula in structural form, we deploy the weak simplification of Definition 16, *without* the short circuiting criteria, such that *true* and *false* can and always will manifest as genuine operands of a disjunction or conjunction alike, after for example acquiring a formula from virtual substitution. This is "simplification preserving structural form". The purpose of structural form is entirely to enable a one to one correspondence between a formula at a particular atomic position of the unquantified part of input Φ to the (successive) results of virtual substitutions at the formula for an `IQER`. Structural form for formulae rely entirely on being locally canonical (Definition 21). "Local" here then means the structure of the input formula, which is deemed to be prenex (only operators are `And` and `Or`), but not necessarily something such as disjunctive or conjunctive normal form.

2.5 Production of Witnesses for QE via VTS

We discuss the production of witnesses for QE problems. Much of this is made rigorous, or extended from [44] and [43].

Definition 22 (Witnesses). *A set of* witnesses *for a homogeneously quantified QE problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$, $Q \in \{\exists, \forall\}$ is a set of assignments of all quantified variables $x_{n-m+1}, \ldots, x_n$ to real numbers $r_{n-m+1}, \ldots, r_n$ that prove equivalence of (1.1) to a meaningful leaf `IQER`.*

Witnesses provide proof of existence of examples or counterexamples for fully existentially quantified or fully universally quantified formulae respectively. This is the only context in which production of such assignments is particularly useful. In contrast, in terms of what is presented here, neither VTS nor CAD can provide *proof* that an existentially quantified formula is equivalent to *false*, or a universally quantified formula is equivalent to *true*. [1] highlights that proof of these equivalences falls to reliance on completeness of the algorithm. In other words it requires a proof that the substitutions performed were *sufficient* to deduce the quantifier free equivalent of the

formula. However, that work also highlights that other newer algorithms for SMT over the theory of real arithmetic (which can be viewed as fully homogeneously existentially quantified QE problems (Section 1.1.1)) are far more amenable to production of proofs in these contexts.

First, we present a fairly simple algorithm that will be necessary in processing witnesses featuring $\pm\infty$, where it suffices to replace these with a real number exceeding a bound for the largest root of any such polynomial in the back substituted formula. This works recursively on the structure of Tarski formulae, as many algorithms will similarly. The ideas owe to [44, 43], and the root bounds themselves are via Cauchy's inequality.

Algorithm 2 Cauchy Root Bound

Input: Ψ, a Tarski formula
Output: c, a rational number representing the maximum Cauchy root bound for all polynomials appearing in Ψ

1: **procedure** CauchyRootBound(Ψ)
2: **if** Ψ is a relation $f \rho 0$ **then**
3: Let $a_1 \dots a_n$ be the integer coefficients of f, with f viewed as a distributed polynomial under any ordering of variables
4: $c \leftarrow 1 + \max_{i=1}^{n} \frac{|a_i|}{|a_n|}$
5: **elseif** $\Psi = \mathit{true}$ or false **then**
6: $c \leftarrow 0$
7: **else**
8: $c \leftarrow \max_{k=1}^{m}$ CauchyRootBound(Ψ_k), where m is the number of operands of Ψ in terms of its outer boolean operator $\ltimes$, and Ψ_k is the kth operand of Ψ viewed this way
9: **end if**
10: **return** c
11: **end procedure**

In particular, the maxima of all bounds for roots of any polynomial occurring in Ψ suffices as a bound for roots of all polynomials occurring in Ψ. Note that any distributed variable ordering would do - in reality we just need all integer coefficients of all such monomials in f as long as f is expanded, so discussing which is meaningless. The check for any atom being *true* or *false* should largely be superfluous, as this algorithm should only be called on (weak) simplified input.

Next is a presentation of an algorithm that allows us to process prewitnesses obtained from test points used in VTS to real numbers in the case where Φ is a fully quantified formula. As such we obtain witnesses as per Definition 22. Again, the ideas are from [44, 43], but this is the canonicalization as an algorithm with `IQER`s, and is implemented in `QuantifierElimination`. The initial presentation of the algorithm was via [64].

Algorithm 3 Production of Witnesses from a VTS `IQER`

Input: L, a meaningful leaf `IQER` for a homogeneously quantified problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$, $Q \in \{\exists, \forall\}$
Output: A list of processed witnesses $[x_1 = r_1, \ldots, x_n = r_n]$, where $r_i \in \mathbb{R}$, $i = 1, \ldots, n$, and Φ evaluated at $[x_1 = r_1, \ldots, x_n = r_n]$ is equivalent to the meaningful truth value associated with L

1: **procedure** GetWitnessesIQER(L, $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$)
2: witnesses $\leftarrow$ an empty `Array`
3: **for** i **from** $L \mapsto$ level $+ 1$ **to** n **do**
4: Append $x_{n-i+1} = 0$ to witnesses
5: **end for**
6: $I \leftarrow L$
7: **while** I has a parent `IQER` **do**
8: Let F be the quantifier free formula associated to I ($I \mapsto$ formulaSimplified)
9: $T \leftarrow I \mapsto$ testpoint evaluated at all current witnesses ▷ Making the polynomial in T univariate
10: $I \leftarrow I \mapsto$ parent
11: Construct the "prewitness" or "root description" associated to T as $x_{n-I\mapsto\text{level}+1} = t$, where t may feature ε or ∞
12: **if** $t = \pm\infty$ **then**
13: Let F_{parent} be the quantifier free formula associated to I, evaluated at all current witnesses
14: Add $x = \text{sgn}(t)$ CauchyRootBound(F_{parent}) to witnesses
15: **elseif** $t = r \pm \varepsilon$ **then**
16: Let F_{parent} be the quantifier free formula associated to I, evaluated at all current witnesses and unsimplified
17: Let rootList be a complete ordered list of isolating intervals of all real roots in $x_{n-I\mapsto\text{level}+1}$ for all polynomials in F_{parent} such that rootList $= [[a_1, b_1], \ldots, [a_k, b_d]]$, $d > 0$ ▷ Using Algorithms 18 and 6, after decomposition of F_{parent} as a flat set of polynomials
18: Let $b_0 = -\infty$ and $a_{k+1} = \infty$
19: **if** $d = 0$ **then**
20: $r \leftarrow 0$ ▷ F_{parent} equivalent to *true* (existential case) or *false* (universal case)
21: **elseif** $t = r - \varepsilon$ **then**
22: **for** j **to** $d + 1$ **do**
23: $s \leftarrow$ the simplest rational strictly between b_{d-j} and a_{d-j+1}
24: **if** $s < r$ **then**
25: $r \leftarrow$ the simplest rational strictly between s and r
26: **break**
27: **end if**
28: **end for**

Algorithm 3 VTS Witness Production Algorithm, Part 2

```
29:             else                                                  ▷ t = r + ε
30:                 for j to d + 1 do
31:                     s ← the simplest rational strictly between a_{j−1} and b_j
32:                     if s < r then
33:                         r ← the simplest rational strictly between r and s
34:                         break
35:                     end if
36:                 end for
37:             end if
38:             Add x = r to witnesses
39:         else
40:             Add x = t to witnesses
41:         end if
42:     end while
43:     return witnesses as a list
44: end procedure
```

Algorithm 3 is essentially a recursive back substitution process — if we can generate a witness for x_i, then we can use it in back substitution to generate a witness for x_{i-1}. It relies, not unlike the methodology of lifting in CAD, on having univariate polynomials available at any one level such that prewitnesses involving non standard symbols ∞ or ε to be converted to witnesses with real numbers. In contrast, any other prewitnesses can always be processed to receive a real number assuming the availability of preceding witnesses. Hence the situations in which witnesses can reliably be generated depend on whether the number of variables in an `IQER`'s formula correspond directly with its level for every `IQER` on the path from that leaf to the root `IQER`, where infinitesimals are contained in any of the test points on that path. On the other hand, if no infinitesimals were used on that path, then processing the prewitnesses on that path is largely a formality, because Maple's eval['recurse'] would successfully use the list of prewitnesses as what would be the corresponding witnesses for $x_{n-m+1}, \ldots, x_n$ by doing the back substitution itself.

In contrast to the initial presentation in [64], Algorithm 3 suggests calling CauchyRootBound to process prewitnesses involving $\pm\infty$ on the formula from the `IQER` above, rather than on the top level quantifier free part of quantified input Φ. Hence back substitution is now relevant on all prewitnesses involving infinitesimals rather than just ε. Additionally, the loop on line 3 suggests to evaluate at all quantified variables that were not eliminated by virtual substitution to receive this leaf node. This implies that a quantifier free equivalent could be deduced without their non trivial elimination, but it is not necessarily the case that the formulae for all `IQER`s on the path to the root node are free of them. Hence we can evaluate at a trivial value, such as $x_i = 0$, to ensure that we have univariate formulae at the appropriate times.

$\Phi(x_1, \ldots, x_n)$ should certainly be homogeneously quantified (quantified with only one type of quantifier) in order for usage of Algorithm 3 to be correct, such that the elimination corresponds to exactly one tree to traverse (Figure 2-2 or 2-3, and not Figure 2-1). `QF_NRA` is one context where the formula is (fully) homogeneously quantified (and the QE is achieved by incrementality on clauses). The algorithm only produces witnesses for meaningful truth values, i.e. *true* for existential questions and *false* for universal questions, and [1] highlights that VTS (and CAD) have similar shortcomings in producing proof of non meaningful truth values for each type of quantification, because it requires proof that the (virtual) substitution points used are a comprehensive covering of meaningful ones.

Theorem 23. *Algorithm 3 is valid, i.e. given a meaningful leaf* `IQER` *L, it correctly produces a list of witnesses $[x_1 = r_1, \ldots, x_n = r_n]$ such that $\Phi(x_1, \ldots, x_n)$ evaluated at $[x_1 = r_1, \ldots, x_n = r_n]$ is equivalent to the meaningful truth value associated to L via its simplified Tarski formula.*

Proof. L need only be a meaningful leaf, and not an `IQER` of level n. If it is a meaningful leaf of level $0 < k < n$, then this implies quantifier elimination was achieved on elimination of Qx_{n-k+1}, and both the the free variables $x_1, \ldots, x_{n-m}$ and the quantified variables $x_{n-m+1} \ldots x_{n-k}$ were superfluous. The loop on line 3 ensures that we always evaluate $x_1, \ldots, x_{n-k}$ at the trivial value 0 in order to eliminate them and coerce the "triangular" condition used extensively below. In other words, in this case it is sufficient to truncate the formula to $Qx_{n-k+1} \ldots Qx_n \; \Phi'(x_{n-k+1}, \ldots, x_n)$ to consider with L', where Φ' and L' are Φ and L evaluated at $[x_1 = 0, \ldots, x_{n-k} = 0]$.

We can proceed by induction on the level of the `IQER` I handled by Algorithm 3 (as of its value per loop iteration on line 8). One notes that $I \mapsto$ level strictly decreases towards 1 per iteration, and the number of variables in the formula for I increases by at most one per iteration. The base case is when I naturally has a univariate formula, i.e. when $I = L$ and any variables superfluous to obtaining *true*/*false* have been evaluated at zero via the loop on line 3. Otherwise we assume Algorithm 3 produces valid witnesses for $x_n, \ldots, x_{n-k+1}$, for some $1 \leq k < n$, and the back substitution from line 9 allows us to view the formula that VTS would have acted upon with hindsight of the values of later quantified variables.

We know that the guard for each test point on the path from L to the root is not equivalent to *false* after back substitution, else L would fail to be a meaningful leaf. Hence T always represents a valid root description. It remains to prove the validity of processing of test points including non standard symbols:

- Handling of prewitnesses involving $\pm\varepsilon$ (line 15):
  - We choose the first value s such that $s < r$ (for $t = r - \varepsilon$) or $s > r$ (for $t = r + \varepsilon$). This is such that the relation from T (after back substitution) is equivalent to a meaningful truth value, but no other polynomials from any other relations change sign due to crossing the boundary of an isolating

interval for x_{n-k+1}, and hence their relations are guaranteed not to change truth value. In the terminology of CAD (which has relevancy due to usage of real root isolation), we essentially select the sample point from the cell from a CAD in $\mathbb{R}$ of F_{parent} such that $x_{n-k+1} = r \pm \varepsilon$ is the meaningful truth value from L for some $\varepsilon > 0$, where the value of ϵ is implied by the relevant loop from the algorithm but not explicitly deduced — ϵ is such that r is the simplest rational, e.g. in terms of having the smallest possible diadic denominator.

- If we reach line 20, then F_{parent} after back substitution should be genuinely equivalent to the meaningful truth value for L. We do not simplify F_{parent} under normal circumstances, but it may require a strong simplifier to deduce that F_{parent} is equivalent to a meaningful truth value, considering e.g. an expression such as $x_{n-k+1}^2 + 1 > 0$ which donates no real roots would potentially yield *true* under strong but not weak simplification. If F_{parent} yields no real roots, we know the signs of the polynomials within are always static, and hence the truth values of the relations are static. Note F_{parent} may even have no polynomials of non trivial degree after back substitution. Because of the assumption that past back substitutions were correct, we know it is *true* if the formula is quantified with existential quantifiers, or *false* in the universal case.

- Handling of prewitnesses involving $\pm\infty$ (line 12): the maxima of the Cauchy root bounds of all polynomials contained in F_{parent} is guaranteed to exceed the modulus of all roots of all polynomials from F_{parent}. We multiply this by the sign of ∞ to give the resulting witness the correct sign.

- Any other prewitnesses are real expressions in $x_n, \ldots, x_{n-k+1}$, and the back substitution converts them directly to a single real number.

Algorithm 3 suggests a tree traversal from leaves to the root `IQER`, unlike various other tree traversal algorithms in this thesis that traverse from the root towards the leaves. Considering the algorithm suggests usage of root isolation per any node involving prewitnesses with infinitesimals, one may worry about the complexity of the process. But we only ever need only process one leaf node via Algorithm 3 — receiving one meaningful leaf `IQER` is sufficient to deduce termination of QE. Therefore we need only traverse the path from one meaningful leaf to the root to provide witnesses under normal termination criteria.

Open Problem 24 (Improving Root Isolation to Process ε). *Can the efficiency of processing of prewitnesses involving ε be improved, either by:*

- *Isolating on fewer polynomials than ALL occurring in the formula — strictly, we need only know about the two closest roots to r (line 21) to find a suitable rational number to replace $r \pm \varepsilon$ by. Currently all polynomials from the* unsimplified *formula F_{parent} are included into the set to isolate, because choice of a value just*

smaller than $r(-\varepsilon)$ or just larger than $r(+\varepsilon)$ and no other roots guarantees that F_{parent} is satisfiable, because we can guarantee the sign of no other polynomials from F_{parent} change.

- *Could usage of root bounds or bounds for root separations help?*

Section 4.4.1 describes how VTS witnesses can be concatenated with those arising from CAD when this project's "poly-algorithm" is used to coerce the "triangularity" condition for an ineligible `IQER` actually equivalent to a meaningful truth value, deduced via CAD.

One notes that as opposed to production of witnesses for a *fully* quantified formula, Algorithm 3 can produce witnesses for meaningful leaves produced for homogeneously quantified formulae, which need not be fully quantified. In this context, the witnesses yielded from usage of the loop on line 3 are merely produced to make usage of is and eval in usage of witnesses canonical for the user, although strictly only the rest of the witnesses are those needed to prove the equivalence of the formula for QE to a meaningful truth value.

2.6 Propagation of VTS

Code Fragment 4 describes "propagation" of VTS via selection of an `IQER`, variable strategy in terms of what was selected, and then construction of a child `IQER` below it using a selected test point.

Fragment 4 Propagation of VTS

1: **while** ($i \leftarrow$ VTSIQERSelectionStrategy(iqers)) > 0 **do** ▷ Section 2.3.1
2: VTSVariableStrategy(iqers[i], vars) to sort variables in vars beyond the maximum level of any `IQER` in the tree ▷ Section 2.3.2
3: Construct *one* child `IQER` I below iqers[i] using Algorithm 1, and test point defined by strategy in getNextTestpoint ▷ Section 2.3.3
4: Add I to iqers
5: **if** If the set of future test points for iqers[i] is empty **then**
6: Remove iqers[i] from iqers
7: **end if**
8: **if** I holds a meaningful truth value via its simplified formula **then**
9: Remove all `IQER`s from iqers
10: Add I to leaves
11: **elseif** $I \mapsto$ level $= m$ **then**
12: Add I to leaves
13: **end if**
14: **end while** ▷ If $i = 0$, QE has been deduced; if $i = -1$, there exist remaining ineligible `IQER`s, and we proceed with CAD in some sense to complete QE (Chapter 4)

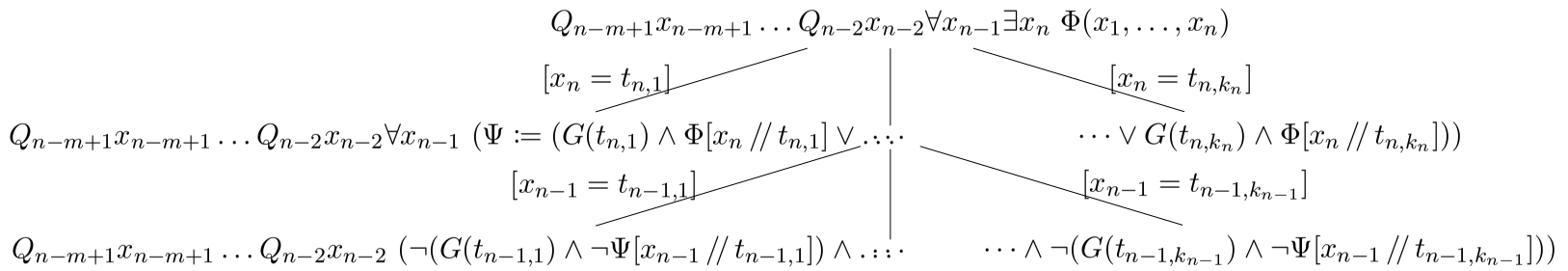
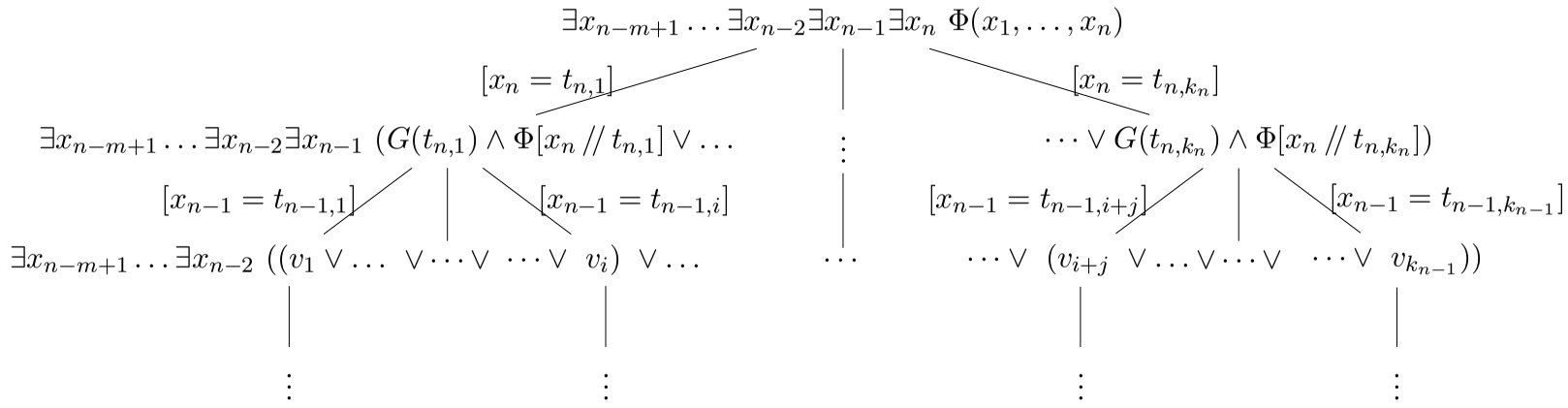
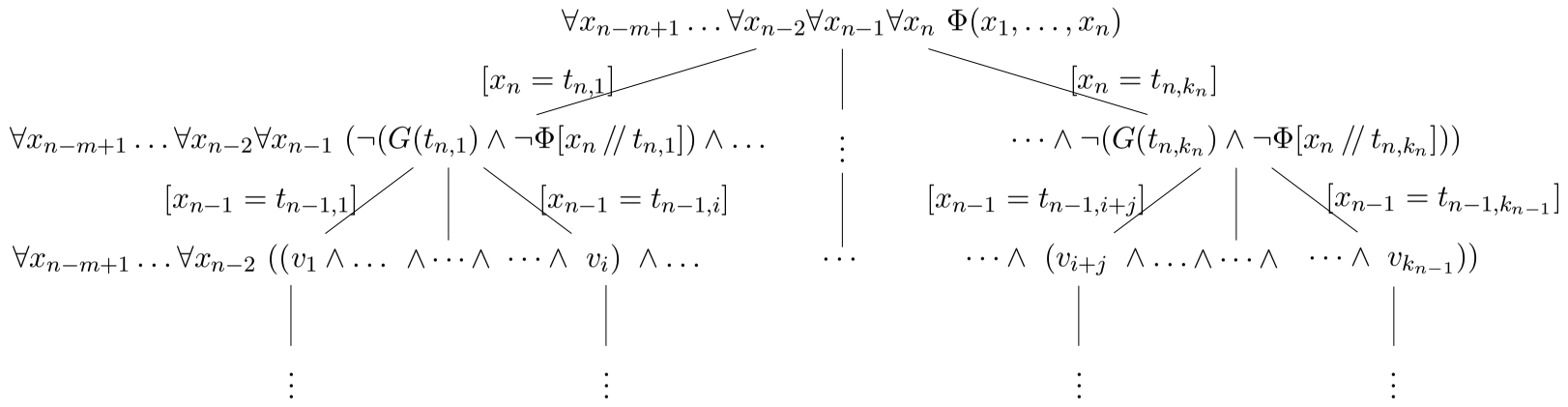


Figure 2-1: The generic layered VTS tree formed by QE with quantifier alternations on (1.1), where the last two quantifiers are forced as $\forall$ and $\exists$ to demonstrate the consolidation of the disjunction formed by elimination of $\exists x_n$ into one implicitly universally quantified formula, Ψ for VTS to traverse next.



Where $v_r = \begin{cases} G(t_{n-1,r}) \wedge (G(t_{n,1}) \wedge \Phi[x_n \,//\, t_{n,1}])[x_{n-1} \,//\, t_{n-1,r}] & r \in \{1, \ldots, i\} \\ G(t_{n-1,r}) \wedge (G(t_{n,k_n}) \wedge \Phi[x_n \,//\, t_{n,k_n}])[x_{n-1} \,//\, t_{n-1,r}] & r \in \{i+j, \ldots, k_{n-1}\} \end{cases}$

Figure 2-2: The VTS tree formed via elimination of a block of existential quantifiers $\exists x_{n-m+1} \ldots \exists x_n$.



Where $v_r = \begin{cases} \neg(G(t_{n-1,r}) \wedge \neg(G(t_{n,1}) \wedge \neg\Phi[x_n /\!/ t_{n,1}])[x_{n-1} /\!/ t_{n-1,r}]) & r \in \{1, \ldots, i\} \\ \neg(G(t_{n-1,r}) \wedge \neg(G(t_{n,k_n}) \wedge \neg\Phi[x_n /\!/ t_{n,k_n}])[x_{n-1} /\!/ t_{n-1,r}]) & r \in \{i+j, \ldots, k_{n-1}\} \end{cases}$

Figure 2-3: The VTS tree formed via elimination of a block of universal quantifiers $\forall x_{n-m+1} \ldots \forall x_n$.

```
> expr := exists( [ x, y, z ], And( x - y = z, y = 5 ) );
```

$$expr := (\exists x)\,(\exists y)\,(\exists z)\ x - y = z \wedge y = 5$$

```
> ( e, q ) := QuantifierEliminate( expr, ProcessWitnesses = false );
```

$$e,\, q := [[true, z = x - y, y = 5, x = 0]],\, true$$

```
> uqf := op( 2, expr );
```

$$uqf := x - y = z \wedge y = 5$$

```
> eval[ 'recurse' ]( uqf, e[1][ 2 .. -1 ] );
```

$$-5 = -5 \wedge 5 = 5$$

```
> ( e, q ) := QE( expr, ProcessWitnesses = true );
```

$$e,\, q := [[true, z = -5, y = 5, x = 0]],\, true$$

Figure 2-4: Example showing that when unprocessed witnesses (i.e. prewitnesses) are returned that do not contain infinitesimals, eval['recurse'] can still be used to prove the equivalence.

```
> expr := QEExamples[ 'Sharir 3-Cube' ];
```

$$expr := (\exists x_1)(\exists x_2)(\exists x_3)\ -25 \le -2x_1 - 25x_2 + 10x_3 \wedge 2 \le 25x_1 + 2x_2 + 10x_3 \wedge -25 \le -2x_1 + 25x_2 + 10x_3 \wedge 2 \le 25x_1 - 2x_2 + 10x_3 \wedge 0 \le -x_2 - x_3 \wedge -2 \le -x_2 + x_3$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses = false' );
```

$$e,\, q := [[true, x_3 = -\tfrac{5}{2} + \tfrac{x_1}{5} + \tfrac{5x_2}{2} + \varepsilon, x_2 = \tfrac{1}{3} - \tfrac{2x_1}{15} + \varepsilon, x_1 = \tfrac{290}{359}]],\, true$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses = true' );
```

$$e,\, q := [[true, x_3 = -\tfrac{4}{3}, x_2 = \tfrac{1}{3}, x_1 = \tfrac{290}{359}]],\, true$$

```
> uqf := op( 2, expr );
```

$$uqf := -25 \le -2x_1 - 25x_2 + 10x_3 \wedge 2 \le 25x_1 + 2x_2 + 10x_3 \wedge -25 \le -2x_1 + 25x_2 + 10x_3 \wedge 2 \le 25x_1 - 2x_2 + 10x_3 \wedge 0 \le -x_2 - x_3 \wedge -2 \le -x_2 + x_3$$

```
> map( is, eval( uqf, e[ 1 ][ 2 .. -1 ] ) );
```

$$true \wedge true \wedge true \wedge true \wedge true \wedge true$$

```
> expr := BuildEconomicsQEExample( 1, 'example' );
```

$$expr := (\exists v_1)(\exists v_2)(\exists v_3)(\exists v_4)(\exists v_5)(\exists v_6)(\exists v_7)(\exists v_8)\ v_7 < 0 \wedge 0 < v_8 \wedge 0 < v_4 \wedge v_2 v_6 + v_3 v_8 = v_4 \wedge v_1 v_5 + v_3 v_7 = v_4 \wedge v_6 = 1 \wedge v_5 = 1 \wedge 0 < v_1$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses = false' );
```

$$e,\, q := [[true, v_8 = -\tfrac{v_2 v_6 - v_4}{v_3}, v_7 = -\tfrac{v_1 v_5 - v_4}{v_3}, v_6 = 1, v_5 = 1, v_4 = v_1 + \varepsilon, v_3 = -\infty, v_2 = v_1 + \varepsilon, v_1 = \varepsilon]],\, true$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses = true' );
```

$$e,\, q := [[true, v_8 = \tfrac{1}{21}, v_7 = -\tfrac{1}{28}, v_6 = 1, v_5 = 1, v_4 = \tfrac{4}{7}, v_3 = -2, v_2 = \tfrac{2}{3}, v_1 = \tfrac{1}{2}]],\, true$$

```
> uqf := op( 2, expr );
```

$$uqf := v_7 < 0 \wedge 0 < v_8 \wedge 0 < v_4 \wedge v_2 v_6 + v_3 v_8 = v_4 \wedge v_1 v_5 + v_3 v_7 = v_4 \wedge v_6 = 1 \wedge v_5 = 1 \wedge 0 < v_1$$

```
> map( is, eval( uqf, e[ 1 ][ 2 .. -1 ] ) );
```

$$true \wedge true \wedge true \wedge true \wedge true \wedge true \wedge true \wedge true$$

Figure 2-5: More nuanced examples of processed VTS witnesses, and how the output from QuantifierElimination can be used to verify the equivalence of existentially quantified formulae to *true* via evaluation of the unquantified part of a formula at the actual list of witnesses.

Chapter 3

Cylindrical Algebraic Decomposition

Cylindrical Algebraic Decomposition (CAD) is perhaps the most well known and most commonly implemented method to solve QE over the reals. CAD's genesis dates back to 1975 via Collins [18], who first detailed building a CAD via *projection* and *lifting* stages.

3.1 Background

A *Cylindrical Algebraic Decomposition* is a decomposition of $\mathbb{R}^n$ into a finite set of connected disjoint regions called *cells*. Cells are semi-algebraic sets, in the sense that they can be described by conjunctions of relations of the form $f \, \rho \, 0$ where $\rho \in \{<, =\}$ and $f \in \mathbb{R}[x_1, \ldots, x_n]$. CADs are algebraic because every cell can be described by a semi-algebraic set. The *cylindricity* property requires that for any two cells c_1 and c_2, their projections onto real space corresponding to $x_1, \ldots, x_k$ where $k < n - M$ (where $1 \leq M \leq n$ is the maximum level of either of these cells) are either identical or disjoint. The cylindricity property is what enables Quantifier Elimination by CAD, in conjunction with the fact that we require that every cell is (at a minimum) sign invariant on all polynomials from an input formula for QE, because sign invariance of all polynomials from input allows us to achieve truth invariance on all relations from input for every cell.

"Projection and lifting" is a common methodology for constructing a CAD, and was the methodology of the original CAD via Collins. That being said, it is not the only methodology, such as the methodology of `RegularChains` which uses triangular decompositions to construct a CAD in complex space, before conversion to one in real space. We discuss background and define further terms for a projection and lifting CAD.

Projection and lifting algorithms to generate CADs are usually seen as proceeding in exactly the two steps projection and lifting. Projection is the process of generating (at least square-free, but here fully factored) bases in successively fewer variables to obtain "projection bases[1] of all orders", or "...of all levels". The following definitions

[1]Sometimes called "projection factors" in other literature, to reflect that the bases are at least factored to be square-free.

are only intelligible in terms of a fixed variable ordering $x_1, \ldots, x_n$ — one notes that a variable ordering for CAD is fixed *at latest* by the selection of a set of projection bases of all orders before lifting (Section 3.8).

Definition 25 (Projection Operator). *A* projection operator *is a function* $P : \mathbb{R}[x_1, \ldots, x_i] \to \mathbb{R}[x_1, \ldots, x_{i-1}]$ *for some* $i \in \{2 \ldots n\}$*, such that if* $A \mapsto B$ *under* P*,* B *having some property* Z *implies the same property* Z *for* A*.*

Definition 26 (Polynomial Level). *The* level of a polynomial f *in CAD is defined as the greatest* $1 \leq i \leq n$ *such that* $f \in \mathbb{R}[x_1, \ldots, x_i]$*, under a fixed variable ordering* $x_1, \ldots, x_n$*. Similarly, the level of a set of polynomials* B *is the greatest* i *such that* $B \subset \mathbb{R}[x_1, \ldots, x_i]$*, again under a fixed variable ordering* $x_1, \ldots, x_n$*.*

The purpose of projection is to deduce polynomials of all levels necessary to construct a CAD with respect to some invariance property (at the very least usually sign invariance). Typical operations included in a projection operator P are pairwise resultants to deduce "crossings" between polynomials, discriminants on individual polynomials to deduce when the number of roots change, and particular coefficients of individual polynomials again to deduce similar facts. A large majority of research in CAD has focused on projection operators, and various projection operators are listed in Section 3.3. Relatedly, much research has focused on equational constraints in projection (Section 3.7), which further lessens the number of polynomials in projection due to the observation that examination of non zero sign of polynomials from such constraints is meaningless.

On the completion of projection associating a fixed variable ordering $x_1, \ldots, x_n$, one now commences with lifting. Very loosely, lifting is a "back substitution" process according to the real roots of projection polynomials, and points from within open intervals defined by such roots. As such a back substitution process, other literature occasionally identifies the lifting process as being further subdivided into a "base phase" for the first step in creating a decomposition of $\mathbb{R}$ from the level 1 projection polynomials, and then a subsequent "lifting phase" for all other levels.

The base phase examines the projection polynomials of level 1, which are univariate, and constructs a decomposition of $\mathbb{R}^1$ according to their real roots. Due to the invariance of each resulting cell via the invariance property associated with projection, one need only associate a single unique real algebraic number as "sample point" in x_1 to each cell. As the real roots of the univariate polynomials may be irrational, such sample points may be real algebraic numbers (Definition 30). Additionally, one associates a positive integer as "index" to each cell. This index identifies the position of each cell with respect to the increasing real line. As such, these cells line up in terms of increasing cell indices. Those cells representing the "open intervals" between real roots are referred to as "sectors", and have odd indices, while the cells owing to exact real roots are referred to as "sections", with even indices. These sectors have the freedom to associate a rational number as their sample point in x_1, because they represent open intervals.

The lifting phase extends upon the base phase via the readily available cells of level 1 obtained via the base phase, when $n > 1$. This is where the sentiment of "back

substitution" comes into play. Examining the projection bases of level 2, substitution of the sample point of a level 1 cell c into such a basis will yield a univariate set of polynomials in x_2. By isolating real roots on the (fully factored) basis of such polynomials, we can decompose c with respect to x_2 to receive level 2 cells, which have c as their parent cell. As such they are in the cylinder over c. These cells also associate local indices and sample points with respect to x_2. The decomposition of c to further cells of a level higher is referred to as "stack construction". All such cells store "local" information with respect to their level — such as a sample point and index. Cells also store an essentially cosmetic local "cell description" corresponding to their region of real space, with a full cell description acquired similarly to the case for sample points (Cell Descriptions for `CADCells`, Section 3.4). One can define a level 1 cell's local sample point as its full sample point, and the full sample point of any cell as the concatenation of its local sample point with the full sample point of its parent cell. The full index of a cell is defined similarly. With such definitions, one obtains methodology to construct the stack of any cell of level less than n, which defines the lifting process. One need only substitute all the elements of a the full sample point of a level k cell into the projection bases of level $k+1$ to acquire a univariate basis for stack construction. Stack construction is defined by Algorithm 14 (CCHILD). Via cell parenting, we acquire a tree structure on cells.

One can essentially view the base phase as stack construction on the "root cell", which is a cell of level 0, completing the canonicalization of a tree structure for cells. The root cell associates no non trivial sample point or cell index — both of which can be viewed as "empty" to reconcile with the recursive definitions of full sample points and indices. The root cell implicitly represents $\mathbb{R}^n$. In this way, the base phase of CAD can instead be viewed as the first instance of the lifting phase on the root cell.

Definition 27. *The* level *of a cell c is $k > 0$, where k is the level of the projection basis used in its construction, or 0 if c is the root cell. Equivalently, the level of a cell is its level in the CAD tree formed by cell parenting.*

Definition 28. *The* local sample point *of a level $k > 0$ cell c is an equation $x_k = r$ where $r \in \mathbb{R}$ is a real algebraic number (Definition 30). The* full sample point *of a cell is its local sample point if the cell is level 1, else it is the concatenation of its' parent's full sample point with its local sample point.*

Definition 29. *The* local index *of a level $k > 0$ cell c is a positive integer representing its position in the stack over its parent cell with respect to the increasing real line with respect to x_k. A cell's* full index *is its local index if the cell is level 1, else it is the concatenation of its' parent's full index with its local index with.*

With a view to CAD for QE, one notes that the sign invariance of cells on the polynomials from input (in this case the unquantified part of input (1.1) Φ) allows us to achieve truth invariance for each cell with respect to Φ. By having each cell evaluate Φ at its full sample point (a process called evaluation of CAD cells, Section 3.4), we can begin to collect the cells that have determinate truth values with respect to Φ, which allows us to deduce the quantifier free equivalent of (1.1). In fact, stack construction on cells with determinate truth values can already be seen to be superfluous in this

case, and furthermore cells with determinate truth values may allow us to disregard stack construction on other cells in the tree (via propagation of truth values in Algorithm 15, PRPTV). This sentiment is the often mentioned breakthrough "Partial CAD" methodology for QE by CAD, and is discussed in 3.4. Construction of quantifier free equivalents for QE by CAD is provided via full cell descriptions in Section 3.4. `QuantifierElimination`'s CAD implementation is highly object oriented, especially with respect to CAD cells and the "local" data that they associate. Many algorithms, especially those incremental ones in Section 5.2 fall to tree traversal in order to only modify local data of cells, implicitly modifying data for whole subtrees of cells at once.

The original worst case time complexity of Collins' CAD was $\mathcal{O}(d^{2^{2n+8}}m^{2^{n+6}})l^3k$, where n is the number of variables, d the maximum degree of any polynomial in any variable in input, m the number of polynomials occurring in input, k the number of occurrences of polynomials and l the maximum coefficient length. In particular, attention is paid to the fact that CAD is "doubly exponential in the number of variables". One reason for this complexity is attributed to the computation of iteratively nested resultants in the process of full projection. So far no projection operator is to avoid computing such polynomials, but optimisation of the projection operator to make the projection polynomials fewer has constantly been of interest, both via new projection operators, and usage of "equational constraints" to restrict such operators.

While CAD has interest as a "complete" algorithm for QE (being able to traverse a problem of any degree, unlike VTS), it has further uses in unquantified contexts for real algebraic geometry, such as motion planning. This refers to a "full" CAD where every level n leaf cell is present. We may refer to this as "full" or "stock" CAD (Algorithm 38), and contrasts with CADs produced with the intent of QE (Algorithm 37), which are referred to as "Partial CAD", which is the name of the methodology from [36] to provide early termination of lifting in CAD for QE, implemented here. In Partial CAD, one may not construct every level n cell due to determinate truth values on cells of a lower level, making stack construction on such cells superfluous. A subset of the methods used in Partial CAD are used to achieve full CAD, and elements of the methodology differ to the extent both are covered in this section.

One notes that the generality of processing quantifiers as "blocks", i.e. assuming homogeneous blocks of quantifiers as we did in VTS vanishes with CAD — CAD used for QE eliminates all quantifiers including alternations in the same CAD call. In this sense there is no concept of recursion for QE in CAD in the same sense as VTS. Hence we once again consider a generically quantified prenex $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi(x_1,\dots,x_n)$ (1.1).

3.2 Tarski Formulae for CAD

CAD, unlike VTS, can accommodate formulae or sets of polynomials over $\mathbb{R}$, rather than merely $\mathbb{Q}$, due to its action on real space. This extends the scope of input to truly realise the terms "real algebraic geometry" and "Quantifier Elimination over the Reals", allowing for formulae such as $\exists q\ -1 \leq q \wedge q \leq 1 \Rightarrow (525000\,(2+\sqrt{2})u_1 - 525000\,\sqrt{2}u_2 - 525000\,\sqrt{2}v_2 = 0) \wedge (-525000\,\sqrt{2}u_1 + 52500\,\sqrt{2}(q+20)u_2 - 52500\,\sqrt{2}v_2 =$

$0) \wedge -525000\sqrt{2}u_1 - 52500\sqrt{2}u_2 + 52500\sqrt{2}(q+20)v_2 = -10$ [39] which includes the irrational $\sqrt{2}$.

Definition 30 (Real Algebraic Number). *A* Real Algebraic Number *is a rational number, or the real root α of some polynomial p with real algebraic number coefficients. An irrational real algebraic number associates an isolating interval (a, b) where $a, b \in \mathbb{Q}$, $b > a$ such that $a \leq \alpha \leq b$. In Maple, such a real algebraic number α can be represented by the expression* $\text{RootOf}(p, a..b)$*, and furthermore we may use this notation to represent truly irrational real algebraic numbers, i.e. those in $\mathbb{R}\backslash\mathbb{Q}$, once again noting that $\mathbb{R}$ is here identified as the countable set of these real algebraic numbers.*

The Maple implementation of CAD in `QuantifierElimination` uses RootOfs indexed by intervals to represent irrational real algebraic numbers. This means Maple expressions that are equivalent to real algebraic numbers are parsed to this RootOf format under the hood in `QuantifierElimination` when they appear as input to CAD, given CAD can process input over $\mathbb{R}[x_1, \ldots, x_n]$. For example, the radical $\sqrt{2}$ is parsed to $\text{RootOf}(x^2 - 2, a..b)$ for suitable $a, b \in \mathbb{Q}$ such that $0 < a < \sqrt{2} < b$. Parsing of input to be over real algebraic numbers in this format is important for the implementation of CAD, such that projection polynomials are over this expected format, implying that lifting polynomials are also over this format.

Definition 31 (Real Algebraic Function). *A* Real Algebraic Function *is an expression* $\text{RootOf}(p, \text{index} = \text{real}[i])$*, $p \in \mathbb{R}[\_Z, x_1, \ldots, x_n]$, $0 < i \leq \deg_{\_Z}(p)$, or an expression equivalent to it (e.g. after evaluation of the RootOf in Maple).*

By "an expression equivalent to it" in Definition 31, consider that $\text{RootOf}(x\_Z - y, \text{index} = \text{real}[1]) \equiv \frac{y}{x}$ (generally any p linear in $\_Z$ leads to this equivalence & hence evaluation). The attention to $\_Z$ due to Maple's designation that $\_Z$ is the active variable in a RootOf. Real algebraic numbers are a subtype of real algebraic functions. The expression $\text{RootOf}(\_Z^2 - 2, 1..2) = \text{RootOf}(\_Z^2 - 2, \text{index} = \text{real}[2])$ — real algebraic numbers always have an intelligible real index.

A Real Tarski Formula (Definition 32) is merely an extension of a Tarski Formula (Definition 3) to allow for real number coefficients in the polynomials instead of merely integers (or rationals).

Definition 32 (Real Tarski Formula). *A* Real Tarski Formula *(RTF) is a polynomial constraint, i.e. $f \rho\ 0$ where $f \in \mathbb{R}[x_1, \ldots, x_n]$, $n \in \mathbb{N}$, $\rho \in \{<, \leq, \neq, =\}$ or a boolean formula of Real Tarski formulae, where allowable boolean operators may feature $\wedge, \vee, \Rightarrow, \veebar$.*

Definition 33 (Extended Tarski Formula). *An* Extended Tarski Formula *(ETF) is a polynomial constraint $f \rho\ 0$ where f is a polynomial with coefficients as real algebraic functions, and $\rho \in \{<, \leq, \neq, =\}$.*

As real algebraic functions are a supertype of real algebraic numbers, extended Tarski formulae form a supertype of real Tarski formulae, which are themselves clearly a supertype of Tarski formulae. These extensions on Tarski formulae are not dissimilar to those from [12], except for the bespoke requirements and representations of the

real expressions within. The above definitions are each realised by Maple types defined by `QuantifierElimination` — `realalgnum`, and `realalgfun` for real algebraic numbers and functions respectively. Meanwhile, each of the Tarski formulae have a relevant "inert" type e.g. `RTFInert`, but also a parallel type to allow for storage of such a formula in an `Array` to allow for the mutability required to build and modify such formulae. This is very similar to the methodology for storing Tarski formulae for VTS in Section 2.4, and also enables mutability to allow for insertion of formulae as is required to realise that aspect of CAD incrementality (Section 5.2). In particular `QuantifierElimination` desires for all polynomials and relations to be canonical in formulae for CAD, and that polynomials are canonical in any fully factored basis of polynomials, because it is important that we do not duplicate polynomials before attempting expensive operations such as root isolation or similar on them.

Simplification of formulae is less relevant for CAD than VTS, because quantifier free output for QE by CAD is an extended Tarski formula given by (3.1) deduced from the cell descriptions of leaf cells. Despite the fact these formulae can never have inner clauses equivalent to *false*, and so they are normal (and closer to candid) in the sense that CAD can never return something non *false* or non *true* when the input formula is equivalent to *false* or *true* respectively (unlike VTS), these formulae can still be very long and unsimplified. Work in this area such as [16] investigates simplification of output CAD formulae. Meanwhile, the evaluation of the unquantified part of input formula for QE Φ is intrinsically part of the process of Partial CAD, i.e. deduction of when Φ is identically *true* or *false* is somewhat exactly the purpose of evaluation of cells in Partial CAD (Section 3.4). Attempting strong simplification on the real Tarski formula held by a cell may deduce that a cell is a leaf early, except that further lifting on this cell to decompose it has essentially the same purpose — to find determinate truth values amongst those cells to deduce the truth value of the original cell. Hence strong simplification on the formulae held by cells would bizarrely negate the purpose of using CAD for QE in the first place. Altogether, CAD in `QuantifierElimination` only ever uses weak simplification on formulae, including weak structural simplification when simplifying formulae in structural form to achieve incrementality (Section 5.2). Similarly to VTS, it allows for delayed evaluation of substitution on formulae (Section 3.4.2).

3.3 Projection

Projection is the process whereby bases of polynomials are generated in successively fewer variables, where such polynomials are deemed to be sufficient to build a sign invariant CAD, usually implied by other invariance properties from the projection operators used.

Iterating use of a projection operator forms "projection of all orders" or "full projection", such that we eventually receive irreducible bases of polynomials in $\mathbb{R}[x_1, \ldots, x_i]$ for all $1 \leq i \leq n$, or in other words bases of levels 1 to n. From here, a process akin to "back substitution" can commence owing to the real roots (Section 3.4), starting from the univariate polynomials, to produce finer and finer geometry from the roots of the successive projection sets.

Typical operations used in projection are resultants, discriminants, and coefficients. CAD is known to have worst case time complexity doubly exponential in the number of variables, and this can be attributed to the projection process. If the polynomials from the input formula are degree $\mathcal{O}(d)$, then the initial resultants are degree $\mathcal{O}(d^2)$, and iterating this process $n-1$ times makes the process doubly exponential. As a result, much research has focused on improvements to the projection operator used in CAD. In general, such improvements prove that fewer polynomials (via fewer operations) are required to sufficiently produce a sign invariant CAD, which is all that is required for truth invariance of the CAD to solve QE. Even then, resultants are still required, however attention is paid such that they are fewest possible. We enumerate various past and present projection operators defined from previous research.

Collins Projection, P_C [18]

$$P_C(A) = PROJ_1(A) \cup PROJ_2(A)$$

$$PROJ_1(A) = \bigcup_{\substack{F \in A \\ F < G}} \Big(\{ \text{lcoeff}_x(f) \mid f \in F^* \} \cup PSC(F^*, F^{*'}) \Big)$$

$$PROJ_2(A) = \bigcup_{\substack{F,G \in A \\ F < G}} \bigcup_{\substack{F^* \in RED(F) \\ G^* \in RED(G)}} PSC(F^*, G^*)$$

where $PSC(F, G)$ is the set of all principal subresultant coefficients that are non zero between elements of F and G, or more specifically

$$PSC(F, G) = \{ \text{psc}_k(F, G) \mid 0 \leq k < \min(\deg(F), \deg(G)), \text{psc}_k(F, G) \neq 0 \},$$

and $RED(f)$ is the set of all reducta of f that are non zero, or more specifically

$$\{ \text{red}^i(f) \mid 0 \leq i \leq \deg_x(f), \text{red}^i(f) \neq 0 \},$$

and this extends to action on a set such that

$$RED(F) = \{ RED(f) \mid f \in F \}.$$

Hong Projection, $PROJH$ [35]

$$PROJH(A) = PROJ_1(A) \cup PROJ_2^*(A)$$

$$PROJ_2^*(A) = \bigcup_{\substack{F,G \in A \\ F < G}} \bigcup_{F^* \in RED(F)} PSC(F^*, G)$$

Brown Projection, P_B

$$\begin{aligned} P_B(A) = \{ \text{discrim}_x(f) \mid f \in A \} \ &\cup \ \{ \text{res}_x(f, g) \mid f \in A,\, g \in A \setminus \{f\} \} \\ &\cup \ \{ \text{lc}_x(f) \mid f \in A \} \end{aligned}$$

McCallum Projection, P_M [50]

$$P_M(A) = \{\text{coeff}_x(f,i) \mid 0 \leq i \leq \deg_x(f), f \in A\} \cup \{\text{discrim}_x(f) \mid f \in A\} \cup \{\text{res}_x(f,g) \mid f \in A,\, g \in A \setminus \{f\}\}$$

Lazard Projection, PL [46, 51]

$$PL(A) = \{\text{discrim}_x(f) \mid f \in A\} \cup \{\text{res}_x(f,g) \mid f \in A,\, g \in A \setminus \{f\}\} \cup \{\text{lc}_x(f) \mid f \in A\} \cup \{\text{tc}_x(f) \mid f \in A\}$$

Projection bases in `QuantifierElimination` are stored within `projection` objects. Sets of polynomials are `MutableSets`, due to the continued interest in usage of mutable data structures in Maple to minimise costs caused by fragmented memory allocation, such as garbage collection. `projection` objects are useful due to the distinction between three main bases of polynomials at any one level — the pivot set, set of inequalities, and basis of unused equational constraints. (Objects via) modules in Maple allow for bespoke ModuleIterator methods to allow one to easily iterate across one level of a `projection` object as if it were a flat set, despite the polynomials being spread across (up to) three sets. An implementation by object also packages the data nicely, and allows canonical methods for printing and incrementality (Algorithm 50). A `projection` object stores the sets $B_{A_1}, \ldots, B_{A_n}$, $B_{E_1}, \ldots, B_{E_{n-1}}$ and $C_1, \ldots, C_{n-1}$ in `Arrays` inequalities, equations and pivotSets, appearing as properties for a `projection` object. They have n, $n-1$, and $n-1$ elements respectively, due to the fact there is no reason to identify information relating to equational constraints at level 1 (what would displace B_{E_1} and C_1). They are `Arrays` to allow for their potential extension in incrementality. If P is a `projection` object, $P \mapsto$ inequalities contains n `MutableSets`. Each $P \mapsto$ inequalities$[i]$ is B_{A_i}, $1 \leq i \leq n$, a basis of polynomials at each canonical CAD level unrelated to equational constraints. For $P \mapsto$ equations and $P \mapsto$ pivotSets, 0 appears at any index where equational constraints were not used (as the default element initialising `Arrays`). As such, checking if $P \mapsto$ equations$[i]$ (or $P \mapsto$ pivotSets$[i]$), $1 \leq i < n$ is of type `MutableSet` is equivalent to checking if equational constraints were used in projection at level $i+1$. Algorithms 5 and 50 demonstrate how how these sets come to fruition, while Figures 3-1 through 3-6 give a visual representation to individual projection steps & how various sets contribute to the bases stored in projection. Algorithms 30 and 32 also make use of the structure of the passed `projection` object. Algorithms typeset here largely refer to the projection bases in terms of their membership of a `projection` object, but in mathematical discussion we will refer to the bases as their more canonical (and shorter) names such as B_{A_i} rather than $P \mapsto$ equations$[i]$ etc. One notes that Algorithm 5 acts upon an existing `projection` object. A `projection` object is initialised with properties as `Arrays` of zeroes during calling CADChooseVarsProjection, where the `Arrays` equations and pivotSets are of length $n-1$ as opposed to length n (Figure 3-7). The elements of these `Arrays` are 0 whenever equational constraints were not used at that canonical CAD level.

`'UseEquations'` is a keyword option for all top level functions involving CAD in `QuantifierElimination` controlling usage of equational constraints in projection in

terms of allowance of restricted projection operations. It features as an argument to Algorithms 5 and 50. The allowable values are `'UseEquations' = 'none'`, `'single'`, or `'multiple'`. One notes that a `'single'` equational constraint does not mean "a single equational constraint, used anywhere in projection", but strictly an equational constraint used at most for the first projection step, and not otherwise. The symbol `'single'` could feasibly be replaced by `'first'` to have more clear meaning, but `'single'` and `'multiple'` reconcile with the typical terminology used in existing literature on equational constraints, such as [48, 49]. `'multiple'` equational constraints means "equational constraints in (semi-)restricted projection wherever feasible". The meaning of `'none'` is clear.

We discuss some remarks on the implementation of the projection phase, particularly in reference to Algorithm 5.

1. Each check "There exists p in P_E is such that $\deg_x(p) > 0$" is *false* when $P_E = \emptyset$.

2. One notes that bases are stored in such a way that their index in storage is commensurate with their level (Definition 26). Polynomials from CAD input also find themselves in bases appropriate to their level, rather than being stored at level n.

3. The first and last projection steps appear as special cases in order to use restricted rather than semi-restricted projection when $n > 1$ for the first projection step and additionally when $n > 2$ for the last projection step. The loop from 2 to $n - 2$ covers any intermediate steps to attempt to use semi restricted projection when $n > 3$. These steps are all otherwise very similar.

4. Usage of single equational constraints merely has us merging any remaining ECs into the set of inequalities as soon as the first projection step is over (line 32). When we specify to use no equational constraints (`'UseEquations' = 'none'`) then this happens before the preprocessing of P_E by Gröbner bases.

5. Line 10 identifies the case where the Gröbner basis deduces that the equational constraints are in themselves inconsistent — via discussion in Section 3.7.3.

6. Figures 3-1 and 3-2 are relevant to the first step of projection — what is encapsulated by line 21. Figures 3-3 and 3-4 demonstrate the intermediate steps of the loop on line 36. Lastly, figures 3-5 and 3-6 cover the last projection step encapsulated by line 47. In each figure, the intention of the arrows is to demonstrate the contribution of various sets to one another. One must create and store bases for any one level, before projecting with respect to the sets formed, not least because projection is more efficient on bases to minimise degrees.

Irreducible Basis Generation in CAD

Irreducible basis generation in CAD is handled by the functions CADMakeBasis and CADMakeBasisWithEqns. `QuantifierElimination` creates fully factored bases via Maple's Factor rather than merely square-free bases via Sqrfree for various reasons

Algorithm 5 Full projection to define all projection bases with restricted projection operations

Input: A a set of polynomials associated with inequalities from input, E a set of polynomials that are equational constraints from input, vars an `Array` of the variables $x_1, \ldots, x_n$ according to the variable ordering chosen, n the total number of variables, UseGroebner a boolean flag as keyword option dictating whether we preprocess E with Gröbner bases (default *true*), PropagateECs a boolean flag as keyword option dictating whether we propagate equational constraints through projection (default *true*), UseEquations a symbol as keyword option dictating how many uses of equational constraints we allow (`'none'`, `'single'`, `'multiple'`, default `'multiple'`)

Output: P, a `projection` object representing projection bases of all levels for the CAD

1: **procedure** PROJECTALLORDERS(A, E, vars, n, UseGroebner, PropagateECs, UseEquations)
2: (P_A, P_E) $\leftarrow A, E$
3: Let $P \mapsto$ inequalities be an empty `Array` with n elements, and $P \mapsto$ equations, $P \mapsto$ pivotSets empty `Array`s with $n-1$ elements
4: **if** UseEquations = `'none'` **then**
5: $P_A \leftarrow P_A \cup P_E$
6: $P_E \leftarrow \emptyset$
7: **end if**
8: **if** UseGroebner **and** $|P_E| > 0$ **then**
9: $P_E \leftarrow$ equationalConstraintsToGroebner(P_E, vars, n)
10: **if** $P_E = \{p\}$ for some polynomial p, **and** $\deg(p) = 0$ **then** ▷ Most likely $P_E = \{1\}$
11: **for** i **to** n **do**
12: $P \mapsto$ inequalities$[i] \leftarrow \emptyset$
13: **end for**
14: **if** $n > 1$ **then** ▷ To notify that equations were actually used
15: $P \mapsto$ equations$[-1] \leftarrow \emptyset$
16: $P \mapsto$ pivotSet$[-1] \leftarrow \emptyset$
17: **end if**
18: **return** P ▷ Hence returning a `projection` data structure with empty bases
19: **end if**
20: **end if**
21: **if** $n > 1$ **then**
22: $x \leftarrow$ vars$[-1]$ ▷ $x = x_n$
23: **if** There exists p in P_E such that $\deg_x(p) > 0$ **then**
24: ($P \mapsto$ pivotSets$[-1]$, P_E, cont$_E$, piv$_c$) $\leftarrow$ choosePivotSet(P_E, x)
25: ($P \mapsto$ inequalities$[-1]$, cont$_A$, self $\mapsto$ equations$[-1]$) $\leftarrow$ CADMakeBasisWithEqns(P_A, P_E, self $\mapsto$ pivotSets$[-1]$, x)
26: (P_A, P_E) $\leftarrow$ lazardProjectionRestricted($P \mapsto$ pivotSets$[-1]$, P_E, $P \mapsto$ pivotSets$[-1]$, piv$_c$, cont$_A$, cont$_E$, x, PropagateECs)

Algorithm 5 Full projection in CAD, Part 2

27: **else**
28: ($P \mapsto$ inequalities[-1], cont_A) $\leftarrow$ CADMakeBasis(P_A, x_n)
29: $P_A \leftarrow$ lazardProjection($P \mapsto$ inequalities[-1], cont_A, x_n)
30: **end if**
31: **end if**
32: **if** UseEquations = `'single'` **then** ▷ Abandon further usage of ECs
33: $P_A \leftarrow P_A \cup P_E$
34: $P_E \leftarrow \emptyset$
35: **end if**
36: **for** i **from** 2 **to** $n-2$ **do**
37: $x \leftarrow$ vars[$-i$] ▷ $x = x_{n-i+1}$
38: **if** There exists p in P_E such that $\deg_x(p) > 0$ **then**
39: ($P \mapsto$ pivotSets[$-i$], P_E, cont_E, piv_c) $\leftarrow$ choosePivotSet(P_E, x)
40: ($P \mapsto$ inequalities[$-i$], cont_A, self $\mapsto$ equations[$-i$]) $\leftarrow$
CADMakeBasisWithEqns(P_A, P_E, self $\mapsto$ pivotSets[$-i$], x)
41: (P_A, P_E) $\leftarrow$ lazardProjectionSemiRestricted($P \mapsto$ pivotSets[$-i$], P_E,
$P \mapsto$ pivotSets[$-i$], piv_c, cont_A, cont_E, x, PropagateECs)
42: **else**
43: ($P \mapsto$ inequalities[$-i$], cont_A) $\leftarrow$ CADMakeBasis(P_A, x)
44: $P_A \leftarrow$ lazardProjection($P \mapsto$ inequalities[$-i$], cont_A, x)
45: **end if**
46: **end for**
47: **if** $n > 2$ **then**
48: $x \leftarrow$ vars[2] ▷ $x = x_2$
49: **if** There exists p in P_E such that $\deg_x(p) > 0$ **then**
50: ($P \mapsto$ pivotSets[1], P_E, cont_E, piv_c) $\leftarrow$ choosePivotSet(P_E ,x)
51: ($P \mapsto$ inequalities[2], cont_A, self $\mapsto$ equations[1]) $\leftarrow$
CADMakeBasisWithEqns(P_A, P_E, self $\mapsto$ pivotSets[1], x)
52: (P_A, P_E) $\leftarrow$ lazardProjectionRestricted($P \mapsto$ pivotSets[1], P_E,
$P \mapsto$ pivotSets[-1], piv_c, cont_A, cont_E, x, PropagateECs)
53: **else**
54: ($P \mapsto$ inequalities[2], cont_A) $\leftarrow$ CADMakeBasis(P_A, x)
55: $P_A \leftarrow$ lazardProjection($P \mapsto$ inequalities[2], $\text{cont}_A \cup P_E$, x)
56: **end if**
57: **else**
58: $P_A \leftarrow P_A \cup P_E$
59: **end if**
60: ($P \mapsto$ inequalities[1], __) $\leftarrow$ CADMakeBasis(P_A, vars[1]) ▷ vars[1] $= x_1$
61: **return** P
62: **end procedure**

discussed in Remark 34. The distinction of CADMakeBasisWithEqns from CADMakeBasis is that it generates bases for a set of inequality polynomials and ECs taking into account a chosen pivot set — these bases may as well both be made disjoint from the pivot set, and furthermore the basis of inequality polynomials may as well be made disjoint from B_E as well. Because of selection of the pivot set by Algorithm 22 to generate C, all polynomials in E are known to have non trivial degree in x.

Remark 34 (Reasoning for Fully Factored Bases). *The motivation for full factorisation of polynomial bases in* `QuantifierElimination` *is mainly to assist root isolation, where the support for factorisation of polynomials over real algebraic numbers is presently stronger than root isolation for the same. Further, it better lends us towards a canonical form for polynomial bases such that set differences of univariate bases $A \setminus B$ should allow us to discard all real roots from B when inspecting $A \setminus B$. Lastly, generation of fully factored bases allows us to fully make use of a caching approach on the underlying operations of projection (resultants, discriminants) in incremental projection within CAD incrementality (Section 5.2.1).*

One notes that the presence of real algebraic numbers as coefficients for polynomials in CAD somewhat frustrates the ability to have a fully canonical form for polynomials in bases. A real algebraic number in our representation cannot have a canonical form due to differing valid isolating intervals, so any references to canonical form for polynomials are in reality *attempts* to make the polynomial canonical, at the very least by making it monic. In particular Maple does not deduce by default that the coefficients between polynomials are equivalent by checking for intersection of isolating intervals where irrational numbers appear.

Algorithm 6 Irreducible canonical polynomial basis creation with respect to a variable in CAD

Input: A, a set of polynomials associated with inequalities, x a variable
Output: B_A, the fully factored canonical basis of all polynomials in A of non trivial degree in x, and cont_A, factors of all polynomials in A of degree 0 in x

```
1: procedure CADMakeBasis( A, x )
2:     B_A ← ∅
3:     cont_A ← ∅
4:     for b in A do
5:         for f in Factors( b ) do
6:             if deg_x(f) > 0 then
7:                 B_A ← ∪{f'}, where f' is f in canonical form
8:             else
9:                 cont_A ← ∪{f}
10:            end if
11:        end for
12:    end for
13:    return B_A, cont_A
14: end procedure
```

Algorithm 7 Irreducible canonical basis creation in CAD with ECs

Input: A a set of polynomials associated with inequalities, E a set of equational constraints of non trivial degree in x that may owe partial factorisation, x a variable, C a basis of factors of the chosen pivot for (semi-)restricted projection in x

Output: B_A, the fully factored canonical basis of all polynomials in A of non trivial degree in x, and cont_A, factors of all polynomials in A of degree 0 in x, and B_E a fully factored canonical basis of all polynomials in E

```
1: procedure CADMakeBasisWithEqns( A, E, x, C )
2:     B_A ← ∅
3:     B_E ← ∅
4:     cont_A ← ∅
5:     for b in E do
6:         for f in Factors( b ) do
7:             f' ← f in canonical form
8:             if not f' ∈ C then
9:                 B_E ← ∪{f'}
10:            end if
11:        end for
12:    end for
13:    for b in A do
14:        for f in Factors( b ) do
15:            if deg_x(f) > 0 then
16:                f' ← f in canonical form
17:                if not f' ∈ C and not f' ∈ B_E then
18:                    B_A ← ∪{f'}
19:                end if
20:            else
21:                cont_A ← ∪{f'}
22:            end if
23:        end for
24:    end for
25:    return B_A, cont_A, B_E
26: end procedure
```

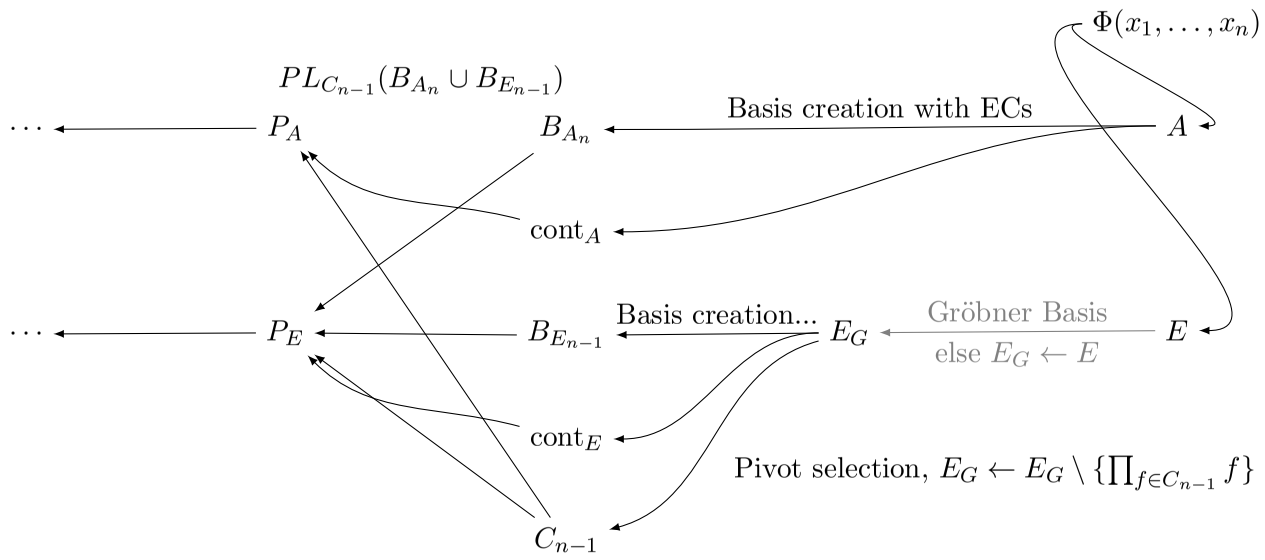


Figure 3-1: Diagram demonstrating initial steps in projection with equational constraints. Φ is decomposed into A and E by Algorithm 21, such that we are able to identify the set of equational constraints E. Further, usage of Gröbner bases to preprocess E is in grey as it is optional depending on the value of UseGroebner. Then E or E_G (and hence $B_{E_{n-1}}$) is assumed to contain at least one polynomial of non trivial degree in x_n such that selection of a pivot set makes sense. The set C_{n-1} is the set of factors of the selected "pivot" from E_G, via Algorithm 22. This pivot is removed from E_G. One then observes usage of restricted Lazard projection via the pivot set C_{n-1} to receive the sets P_A and P_E, where P_E is a set of potential equational constraints for the next level and projection step. P_A and P_E are not necessarily bases, where P_A is a set of polynomials judged to be those from "inequalities", or in other words not equational constraints. The generated P_A and P_E to send to level $n-1$ are $\subset \mathbb{R}[x_1, \ldots, x_{n-1}]$.

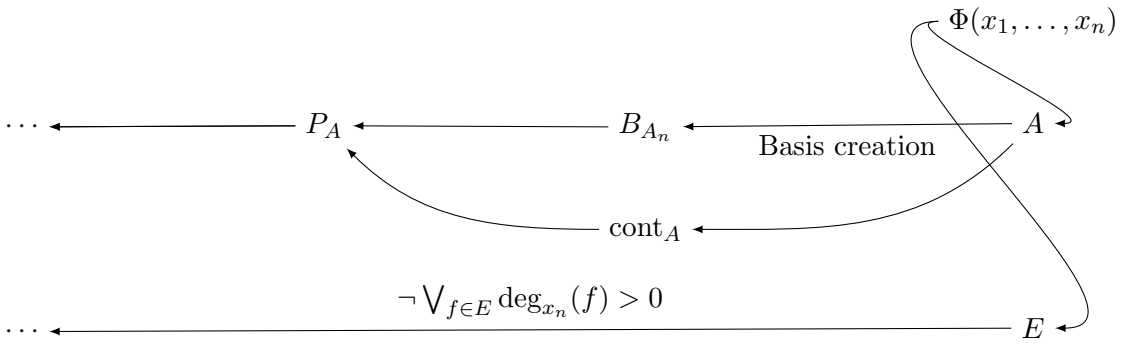


Figure 3-2: Diagram demonstrating initial steps in projection where the set of equational constraints E from input Φ is found to have no polynomials of non trivial degree in x_n (which also includes the possibility that $E = \emptyset$). We use standard Lazard projection on the generated basis set for A, B_{A_n} to receive the set P_A to bring forward to the next projection set. P_E gets carried forward directly through to the next step. The generated P_A and P_E to send to level $n-1$ are $\subset \mathbb{R}[x_1, \ldots, x_{n-1}]$.

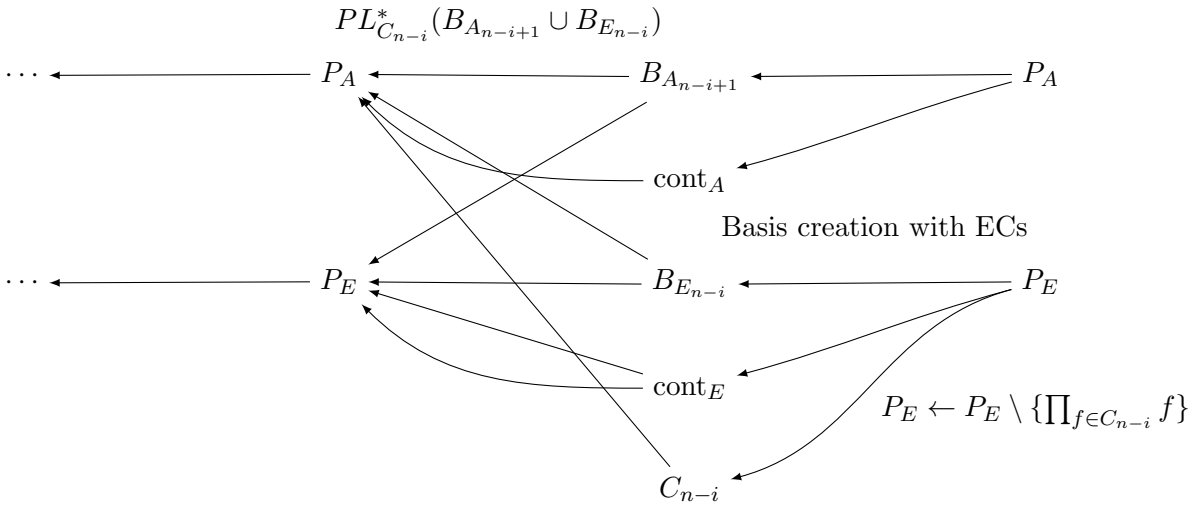


Figure 3-3: Diagram demonstrating an intermediate step in projection with equational constraints. By an intermediate projection step, we mean step i, projection on x_{n-i+1}, for $1 < i < n-1$. P_E (hence also $B_{E_{n-i}}$) is assumed to contain at least one polynomial of non trivial degree in x_{n-i+1} such that selection of a pivot set makes sense. $P_E \leftarrow P_E \setminus \{\prod_{f \in C_{n-i}} f\}$ is the selection of this pivot (Algorithm 22), and hence removal from P_E before generation of the basis $B_{E_{n-i}}$ for P_E. The generated replacement P_A and P_E to send to the next level are $\subset \mathbb{R}[x_1, \ldots, x_{n-i}]$.

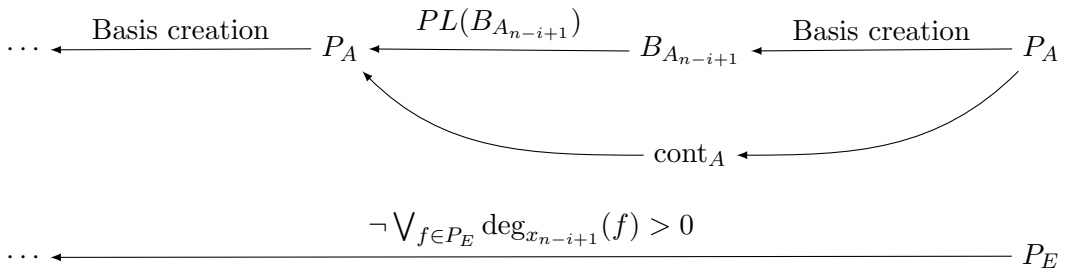


Figure 3-4: Diagram demonstrating an intermediate step in projection, meaning step i to project with respect to x_{n-i+1} where $1 < i < n-1$. This reflects such a step where no available equational constraints exist of non trivial degree in x_{n-i+1}, which also includes the possibility that $P_E = \emptyset$. Hence P_E is brought forward to the next projection step in the next variable. Meanwhile, we perform regular Lazard projection on the basis set generated for P_A, $B_{A_{n-i+1}}$, generating the replacement $P_A \subset \mathbb{R}[x_1, \ldots, x_{n-i}]$ to bring forward to the next level.

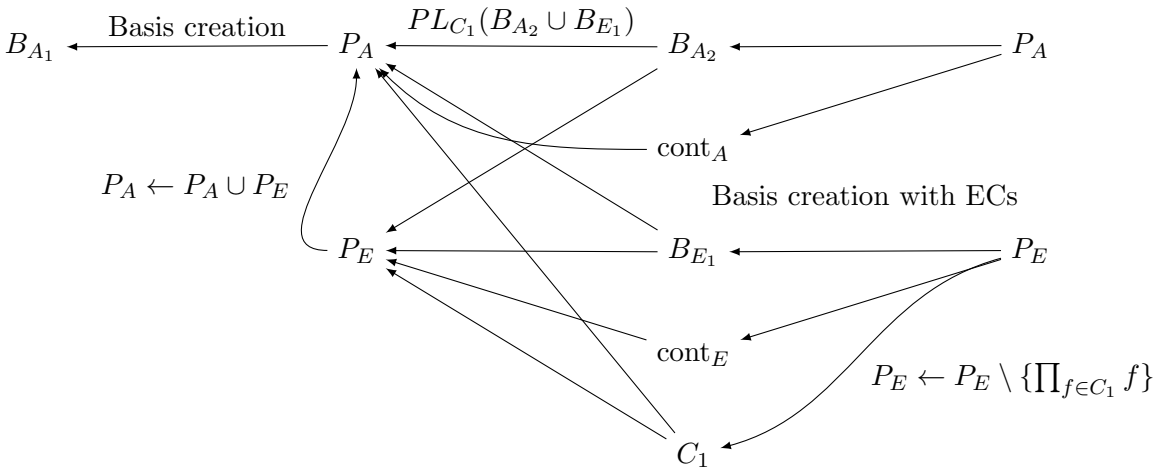


Figure 3-5: Diagram demonstrating the last step of projection with equational constraints. As such P_E is assumed to have polynomials with non trivial degree in x_2, and we use restricted projection on the chosen pivot set C_1 with respect to x_2. The selected pivot (Algorithm 22) must be removed from P_E ($P_E \leftarrow P_E \setminus \{\prod_{f \in C_1} f\}$). This restricted projection produces a P_A and P_E both $\subset \mathbb{R}[x_1]$, on which no projection is meaningful, hence the equational constraints amongst them aren't meaningful, so we merge P_E into P_A. Lastly, we generate the basis on P_A as B_{A_1}, storing these as the univariate polynomials to lift around first.

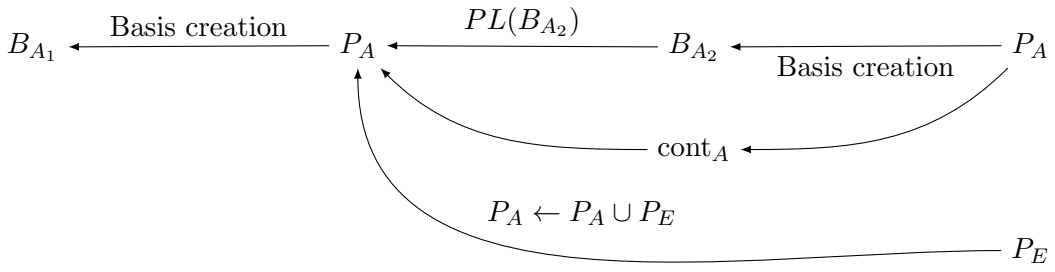


Figure 3-6: Diagram demonstrating last step in projection, where we must have $P_E \subset \mathbb{R}[x_1]$ such that we cannot use equational constraints in the last projection step on x_2. Hence we merely perform standard Lazard projection on B_{A_2} and then as no projection remains to be done, merge in the remaining equational constraints forming a set in $\mathbb{R}[x_1]$. We produce the last basis of inequalities to store, B_{A_1} — storing a basis of equational constraints would be superfluous due to the lack of extra projection to do.

The most contemporary projection operator is the Lazard operator [46, 51]. As a result, this is the operator used in `QuantifierElimination`. The Lazard operator is an improvement on McCallum's by omission of "intermediate" or "middle" coefficients in x, i.e. all but the leading and trailing coefficients.

Other improvements to the projection operator with respect to the context of QE are optimisations in the presence of equational constraints.

3.4 Lifting

Definition 35 (Lazard Delineable). *[51, Definition 2.10] Let $S \subseteq \mathbb{R}^{n-1}$ and $f \in \mathbb{R}[x_1, \ldots, x_n]$. We say that f is Lazard delineable on S if:*

(i) The Lazard valuation of f above β is the same for each point $\beta \in S$.

(ii) There exist finitely many continuous functions $\theta_i : S \to \mathbb{R}$, such that $\theta_1 < \cdots < \theta_k$ if $k > 0$ and such that for all $\beta \in S$, the set of real roots of f_β is $\{\theta_1(\beta), \ldots, \theta_k(\beta)\}$.

(iii) If $k = 0$, then the graph of f does not pass over S. If $k \geq 1$, then there exist positive integers $m_1, \ldots, m_k$ such that, for all $\beta \in S$ and for all $1 \leq i \leq k$, m_i is the multiplicity of $\theta_i(\beta)$ as a root of f_β.

Definition 36 (Lazard Evaluation/Valuation [51]). *The Lazard evaluation of a level $1 \leq k \leq n$ square-free polynomial $f(x_1, \ldots, x_k)$ from projection at the sample point α of a level $k-1$ CAD cell $f_\alpha(x_k)$ is the polynomial returned by the result of Algorithm 11.*

The associated Lazard valuation is $\nu_\alpha(f) = [v_1, \ldots, v_{k-1}] \in \mathbb{N}_0^{k-1}$, where $v_1, \ldots, v_{k-1}$ are defined by the comment on line 8 of Algorithm 11.

Algorithm 8 Lazard projection - implements the operator $PL(A)$

Input: B_A a basis of polynomials all of non trivial degree in x, cont_A set of polynomials free of x, "content" of basis polynomials in B_A, x the variable to do Lazard projection with respect to

Output: P_A, a set of polynomials free of x, not necessarily a square-free basis

```
1:  procedure LAZARDPROJECTION( B_A, cont_A, x )
2:      P_A ← ∅
3:      for i to |B_A| do
4:          f ← B_A[i]
5:          P_A ← P_A ∪ {lc_x(f), tc_x(f), discrim_x(f)}
6:          for j to i − 1 do
7:              P_A ← P_A ∪ {res_x(f, B_A[j])}
8:          end for
9:          for j from i + 1 to |B_A| do
10:             P_A ← P_A ∪ {res_x(f, B_A[j])}
11:         end for
12:     end for
13:     return P_A ∪ cont_A
14: end procedure
```

With Definition 35, we can now understand the property Z as "Lazard delineability" for the Lazard projection operator P_L as P in Definition 25. Lazard evaluation is what enables us to evaluate projection polynomials at sample points to receive univariate lifting polynomials to enable the lifting process. Lazard evaluation differs from standard evaluation in that it avoids the nullification that the McCallum projection operator suffers from (Section 3.7.2). Lazard evaluation/valuation/delineability is also known as lex-least evaluation/valuation/delineability in [57, 56], or in other words "Lazard" is sometimes exchanged for "lex-least" in associated literature, due to properties of Lazard valuations.

Remark 37. *Considering a one to one correspondence between* `CADCells` *and their full sample points, it will be convenient to speak of the Lazard evaluation of a polynomial on a* `CADCell` *or on its sample point as one and the same.*

Definition 38 (Lifting Polynomial). *A* lifting polynomial *is the result of Lazard evaluation of a level $1 \leq k \leq n$ irreducible polynomial at a level $k-1$ CAD cell. The lifting polynomial in this context is hence in $\mathbb{R}[x_k]$, in particular univariate in x_k. The root descriptions of a lifting polynomial appear as bounds for sectors, and as local sample points for local sections created in stack construction (see CCHILD, Algorithm 14).*

Algorithm 12 produces lifting polynomials from projection polynomials via Lazard evaluation, in functions such as CCHILD. The algorithm chooses to fully factor the results of Lazard evaluation, as opposed to merely generating a square-free factorisation — the Maple function Factors is chosen over Sqrfree. This is an implementation choice

Algorithm 9 Restricted Lazard Projection with Equational Constraints — implements $PL_E(A) = PL(E) \cup \{\text{res}_x(f, g) \mid f \in A,\, g \in E \setminus \{f\}\}$

Input: B_A, a basis of polynomials associated with inequalities, E, set of equational constraints of non trivial degree in x that may attribute some partial factorisation each, B_P, basis of factors of the equational constraint chosen as pivot for restricted projection, piv_c the content in x of the chosen pivot for restricted projection, cont_A, set of polynomials free of x, "content" of B_A, cont_E, set of ECs free of x, x the variable to take restricted Lazard projection with respect to, PropagateECs a boolean flag from keyword option dictating whether to propagate equational constraints via the resultant rule

Output: Two sets of polynomials P_A and P_E free of x, not necessarily square-free bases. P_E is a set of polynomials identifiable as ECs in the next variable, and P_A is a set of all other polynomials (implicitly associated with inequalities).

1: **procedure** LAZARDPROJECTIONRESTRICTED(B_A, E, B_P, piv_c, cont_A, cont_E, x, PropagateECs)
2: (P_A, P_E) $\leftarrow \emptyset, \emptyset$
3: **if** PropagateECs **then** ▷ If this option is turned on, we should oblige it by adding resultants from the resultant rule (3.3) to the set of potential pivots for the next projection step
4: $\text{res}_{\text{dest}} \leftarrow P_E$
5: **else**
6: $\text{res}_{\text{dest}} \leftarrow P_A$
7: **end if**
8: **for** g **in** E **do**
9: $\text{res}_{\text{dest}} \leftarrow \text{res}_{\text{dest}} \cup \{\text{piv}_c \prod_{f \in B_P} \text{res}_x(f, g)\}$ ▷ Usage of the resultant rule (3.3)
10: **end for**
11: **if** $\deg(\text{piv}_c) > 0$ **and** ($\text{res}_{\text{dest}} = \{0\}$ **or** $\text{res}_{\text{dest}} = \emptyset$) **then**
12: $P_A \leftarrow P_A \cup \{\text{piv}_c\}$ ▷ No non trivial resultants were computed via usage of the resultant rule in the loop on line 8 so we need to let piv_c manifest as a regular "inequality polynomial" at the top level
13: **end if**
14: **for** i **to** $|B_P|$ **do**
15: $f \leftarrow B_P[i]$
16: $P_A \leftarrow P_A \cup \{\text{res}_x(f, g) \mid g \in B_A\}$
17: **for** j **from** 1 **to** $i - 1$ **do**
18: $P_A \leftarrow P_A \cup \{\text{res}_x(f, B_P[j])\}$
19: **end for**
20: **for** j **from** $i + 1$ **to** $|B_P|$ **do**
21: $P_A \leftarrow P_A \cup \{\text{res}_x(f, B_P[j])\}$
22: **end for**
23: $P_A \leftarrow P_A \cup \{\text{lc}_x(f), \text{tc}_x(f), \text{discrim}_x(f)\}$
24: **end for**
25: **return** $P_A \cup \text{cont}_A$, $P_E \cup \text{cont}_E$
26: **end procedure**

Algorithm 10 Semi Restricted Lazard Projection — implements $PL^*_E(A) = PL_E(A) \cup \{\text{discrim}_x(f) \mid f \in A\}$

Input: B_A, a basis of polynomials associated to inequalities of non trivial degree in x, E, a set of polynomials of non trivial degree in x associated to ECs that may attribute some partial factorisation each, B_P, the basis of factors of the equational constraint chosen as pivot for semi-restricted projection, piv_c, the content of the chosen pivot in x, cont_A, a set of polynomials associated with inequalities free of x, cont_E a set of ECs free of x, x the variable to perform semi-restricted projection with respect to, PropagateECs a boolean flag from keyword option dictating whether we propagate equational constraints through projection

Output: Two sets of polynomials P_A and P_E free of x, not necessarily square-free bases. P_E is a set of polynomials identifiable as ECs in the next variable, and P_A is a set of all other polynomials (implicitly associated with inequalities).

1: **procedure** LAZARDPROJECTIONSEMIRESTRICTED(B_A, E, B_P, piv_c, cont_A, cont_E, x, PropagateECs)
2: (P_A, P_E) ← lazardProjectionRestricted(B_A, E, B_P, piv_c, cont_A, cont_E, x, PropagateECs) ▷ $PL_E(B_A)$
3: **for** f **in** B_A **do**
4: $P_A \leftarrow P_A \cup \{\text{discrim}_x(f)\}$
5: **end for**
6: **for** f **in** E **do** ▷ E is all unused equational constraints, we treat them as inequalities but they are separated for technical reasons
7: $P_A \leftarrow P_A \cup \{\text{discrim}_x(f)\}$
8: **end for**
9: **return** P_A, P_E
10: **end procedure**

Algorithm 11 Lazard Evaluation

Input: c a level $k-1$ `CADCell`, $f \in \mathbb{R}[x_1, \ldots, x_k]$, where $1 \leq k \leq n$, with full sample point α
Output: $f_\alpha(x_k)$, the Lazard evaluation of f at α (Definition 36)

procedure LAZARDEVAL(c, f)
  Let α be the full sample point of c, $[x_1 = a_1, \ldots, x_{k-1} = a_{k-1}]$ where $a_1, \ldots, a_{k-1}$ are real algebraic numbers
  $f_\alpha \leftarrow f$
  for i **to** $k-1$ **do**
    $s \leftarrow \text{lhs}(\alpha_i) - \text{rhs}(\alpha_i)$ ▷ s becomes the ith element of the full sample point as a linear polynomial
    while $s \mid f_\alpha$ **do**
      $f_\alpha \leftarrow \frac{f_\alpha}{s}$
    end while ▷ Let v_i be the number of times $s \mid f_\alpha$ is *true*
    $f_\alpha \leftarrow f_\alpha|_{\alpha_i}$
  end for ▷ It is allowable that $k-1$ can be 0, i.e. c is the root cell, and hence there is nothing to do to $f_\alpha = f$
  return f_α
end procedure

that may differ from other implementations that may merely make square-free bases in general (Remark rmk:fully-factored-reasoning). The purpose of returning the table T is such that lifting polynomials can be linked back to their projection polynomials in order to form real algebraic functions describing the local general cell bounds for new cells (see CCHILD, Algorithm 14).

Definition 39 (`CADCell`)**.** *A `CADCell` object is a member of the CAD tree. As an object, it only stores* local *information via its properties, which include:*

- level : *Its level, i such that $0 \leq i \leq n$ both describing its level in the CAD tree and canonically which variable the CAD cell is associated with locally — x_{n-i+1} if $i > 0$ else the cell is the root cell,*
- sample_point : *Its local sample point — a single equation of the form $x_{n-i+1} = \alpha$ for α some real algebraic number,*
- cell_description : *Its local cell description — a relation $x_{n-i+1}\, \rho\, f$ where f is a real algebraic function on $x_{n-i+2}, \ldots, x_n$ and $\rho \in \{=, <, >\}$, or a conjunction of two strict inequalities $x_{n-i+1}\, \rho\, f$ where f is a real algebraic function on $x_n, \ldots, x_{n-i}$ and $\rho \in \{<, >\}$, or* true *(meaning $-\infty < x_{n-i+1} < \infty$, i.e. x_{n-i+1} can take any value on the real line for this cell) such that this description corresponds to the CAD cell locally in light of the descriptions of its parent(s) in $x_{n-i+2}, \ldots, x_n$,*
- local_index : *Its local index, a positive integer j, even if the cell is a local section, odd if it is a local sector,*

Algorithm 12 Creation of an irreducible canonical basis of univariate lifting polynomials from a set of multivariate projection polynomials

Input: P, a `projection` object, or any other data structure (such as a set) containing polynomials $\subset \mathbb{R}[x_1, \ldots, x_{k+1}]$ that can be iterated over, c a level $k \geq 0$ `CADCell`

Output: B, a fully factored canonical basis of univariate lifting polynomials $\subset \mathbb{R}[x_{k+1}]$, and T, a table mapping polynomials in B to their originating projection polynomial from P

1: **procedure** UNIVARIATEBASISATLAZARD(P, c)
2: **if** P is a `projection` object **then**
3: Set the level for iteration of P as k, such that P iterates over B_{A_k}, $B_{E_{k-1}}$, and C_{k-1}, or B_{A_k} if the latter two sets do not exist
4: **end if** ▷ Else P a set
5: $B \leftarrow \emptyset$
6: **for** b **in** P **do**
7: $b_\alpha \leftarrow$ lazardEval(c, b) ▷ Algorithm 11
8: **for** f **in** Factors(b_α) **do** ▷ Factors(b_α) a list of irreducible factors of b_α without multiplicity
9: $f' \leftarrow f$ in canonical form
10: $B \leftarrow B \cup \{f'\}$
11: $T[f'] \leftarrow b$
12: **end for**
13: **end for**
14: **return** B, T
15: **end procedure**

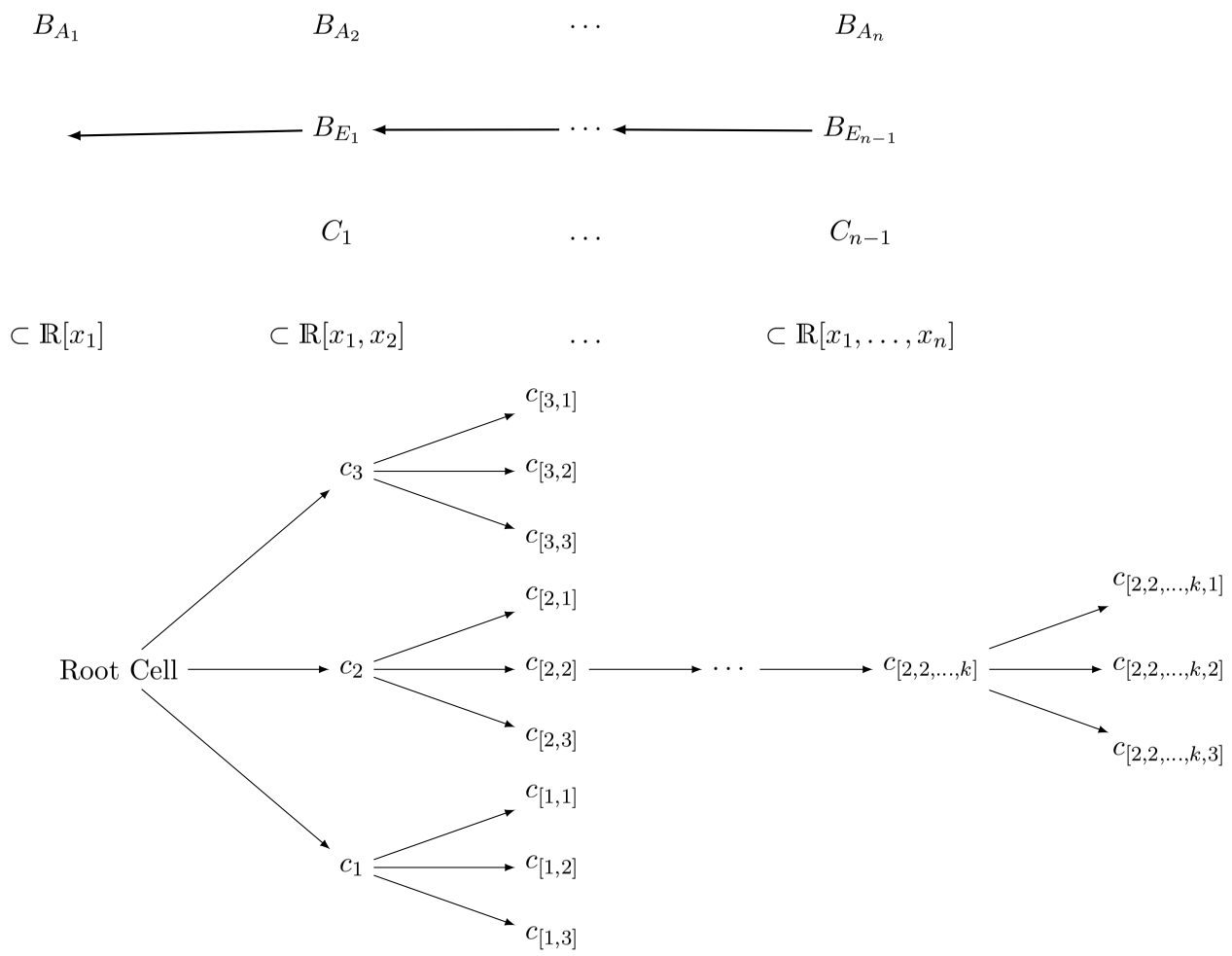


Figure 3-7: A diagram demonstrating the opposition in directions between projection and lifting, including the various projection bases stored by `QuantifierElimination` (B_{A_i}, $1 \leq i \leq n$, and B_{E_i} and C_i, $1 \leq i \leq n-1$). Each CAD cell is vertically aligned with the projection bases it is used with in Lazard valuation to create its child cells. For example, c_1 has a sample point $x_1 = \alpha$, $\alpha \in \mathbb{R}$ to eliminate x_1 in B_{A_2}, B_{E_1} and C_1 to receive univariate polynomials in x_2 to form the local information to create $c_{[1,1]}, c_{[1,2]}, c_{[1,3]}$. The direction of projection is "left", whereas the direction of lifting is "right", with respect to this diagram (via the direction of the arrows). Hence they are in opposition.

- parent : *Its parent* `CADCell`*, if the cell is of positive level, hence forming the structure of the CAD tree,*
- children : *An* `Array` *of its child cells if the cell is of level* $< n$ *and they have been constructed by CCHILD, always sorted with respect to the local indices of those child cells (and hence their alignment from left to right on the real axis),*
- truth_value : *Its truth value (relevant for QE by Partial CAD) — where assigned,* true*,* false*, or the Maple value* FAIL*, all of which type as a boolean value.* FAIL *should be interpreted as "truth value indeterminate",*
- tarski_formula : *Its (real) Tarski formula — where QE is concerned, the equivalent of the unquantified part of prenex QE input* Φ *on the sample point of this cell. The formula is truth invariant over the cell,*
- tarski_formula_structural : *Its (real) Tarski formula in structural form, to enable incrementality by atomic position,*
- lower_bound, upper_bound : *Its lower and upper bounds, mostly relevant where the cell is a local sector — real algebraic numbers obtained as bounds for the cell via its full sample point.*

Algorithm 13 defines the constructor for creation of `CADCell`s. Importantly, creation of the root cell requires slightly different (in particular, fewer) arguments due to the fewer properties to be assigned (much like the constructor for `IQER`s, Algorithm 1). One should pass the unquantified part of quantified prenex input for QE (Φ) to create the root cell such that its Tarski formula is identically Φ when doing Partial CAD, or in full CAD the root cell should receive *true* as the most canonical Tarski formula such that the same number of arguments can be passed — and the tarski_formula is irrelevant (unassigned) for `CADCell`s in full CAD. *true* is also the equivalent Tarski formula to "anywhere in $\mathbb{R}^n$". The root cell never gets assigned further properties except for potentially its truth_value in the context of QE, where propagation of truth values from cells in the tree may allow the root cell to receive a *true* or *false* truth value.

Definition 40 (Local Sector/Section). *A* local sector *is a* `CADCell` *such that the last element of its full cell index (i.e. its local index) is odd. A* local section *is a* `CADCell` *such that the last element of its full cell index (i.e. its local index) is even. In other words,* $c \mapsto \text{local\_index} \bmod 2 = 1$ *or* 0 *for a* `CADCell` c *of positive level to be a local sector or section respectively.*

When defining a local section, the cell does not technically have both a lower and upper bound, due to the cell owing locally to exactly one real algebraic function. Hence 6 arguments are passed for a local section, and the cell only receives a real algebraic number as lowerbound and no upperbound. This lowerbound also easily allows us to deduce the local sample point for a local section, $x_{n-i+1} =$ lowerbound. 7 arguments are required to define a local sector, as the cell requires real algebraic numbers (or $-\infty$, ∞ respectively) for its lower and upper bound.

Algorithm 13 Constructor for a `CADCell`

Input: $\Phi(x_1, \ldots, x_n)$ an unquantified Real Tarski Formula corresponding to top level input, **OR** cd an extended Tarski formula (Definition 33), p the parent `CADCell` for c, lvl the level of the created cell, ind the local index for the created cell, addAsChild a boolean value dictating whether we add this cell as the next possible child for p, lb the lower bound for the created cell (if creating a local sector) or the exact bound for the created cell (if creating a local section), ub the upper bound for the created cell (if creating a local sector)

Output: c, a new `CADCell`

1: **procedure** `CADCell`(cd, p, lvl, ind, addAsChild, lb, ub)
2: **if** 1 argument was passed **then** ▷ Creating the root `CADCell`
3: $c \mapsto$ level $\leftarrow 0$
4: $c \mapsto$ tarski_formula $\leftarrow$ cd ▷ cd $= \Phi$, if doing QE, else cd $=$ *true*
5: **else** ▷ Creating any non trivial `CADCell`
6: $c \mapsto$ parent $\leftarrow p$
7: $c \mapsto$ cell_description $\leftarrow$ cd
8: $c \mapsto$ level $\leftarrow$ lvl
9: $c \mapsto$ local_index $\leftarrow$ ind
10: $c \mapsto$ lowerbound $\leftarrow$ lb
11: **if** addAsChild **then**
12: Add c as the next element of $p \mapsto$ children
13: **end if**
14: **if** 7 arguments were passed **then** ▷ Defining a local sector
15: $c \mapsto$ upperbound $\leftarrow$ ub
16: **end if**
17: **end if**
18: **end procedure**

Local Data for a `CADCell`

One notes that `CADCell`s only store local information, as a matter of storage complexity. Obtaining full information for a `CADCell` is almost always a matter of traversal towards the root cell. For example, a level 1 `CADCell`'s full sample point is the same as its local sample point. For a level $i > 1$ `CADCell`, its full sample point can be found recursively via the concatenation of its parent's full sample point with its local sample point. Usage of a function such as Lazard evaluation (Algorithm 11) requires the full sample point of a cell (represented as a list), hence the method getSamplePoint for a `CADCell` produces this concatenation as a list. For the level 0 root cell, various data is unassigned, such its sample point or index, hence relevant methods attempting to return such data for the root cell return something trivial, such as the empty list for its sample point. Various recursive functions on `CADCell`s make use of the fact that modification of its local data implicitly modifies all cells in the subtree rooted at that cell. Examples of this are Algorithms 52 or 51. Note that one can traverse both up and down the tree, via the parent and children properties respectively. Being able to do both, sometimes in the same algorithm, is a necessity. The subtree rooted at any one cell is in fact the cylinder of that cell, with the leaves of that subtree the finest possible decomposition of that cylinder. In total, while a `CADCell` only stores local information with respect to the variable it was built around, it represents a subset of $\mathbb{R}^n$ in light of the data from its ancestral cells. The child cells of any cell represent a decomposition of that cell with respect to one axis, and this decomposition of $\mathbb{R}^n$ is finer unless there is only one child cell (in which case there were no roots to decompose with respect to, and the decomposition is exactly as fine).

Partial CAD

Partial CAD is a pivotal optimisation for the CAD algorithm provided by Collins & Hong [36] tailored completely to QE. In particular, as opposed to building every single cell, one can terminate building the CAD when a satisfactory cell (or set of cells) has been built that solves the QE problem. As an example, for a fully existentially quantified problem, Partial CAD enables the best case that one need only build one maximum level cell, for which the sample points associated to this cell are witnesses for which substitution into Φ makes Φ equivalent to *true*. Given that PartialCylindricalAlgebraicDecompose within `QuantifierElimination` has the main purpose of QE, it is obvious that the implementation should closely follow the work of Partial CAD.

Algorithms 14 and 15 are based on the "Partial CAD" work from [36]. They describe the creation of child cells (hence CCHILD) and propagation of truth values (hence PRPTV) from those child cells towards the root where possible. In terms of the discussion above, CCHILD creates unevaluated cells, while PRPTV evaluates them. Their formulation here is contextual for `QuantifierElimination`, but each are heavily based on the original formulations from [36]. The early termination of Partial CAD arises from usage of PRPTV, which propagates truth values to the extent that we may be able to identify unevaluated cells as meaningless to evaluate and evaluated cells as meaningless to pass through further stack construction.

Algorithm 14 constructs the stack beneath an evaluated `CADCell` c with a determinate or superfluous truth value in order to gain child cells in terms of the CAD tree. The cells are constructed via the `CADCell` constructor Algorithm 13, with cell bounds deduced via real root isolation of lifting polynomials obtained via Lazard evaluation of projection polynomials of a level appropriate to the cell c.

Some comments on Algorithm 14, CCHILD:

- CCHILD represents stack construction with respect to the next canonical variable for a cell. The children of the cell c are subsets of c in space, in the cylinder of c, and in some sense "replace" c in terms of the canonical container of unevaluated cells in CAD, cad, due to the decomposition.
- Real algebraic numbers & functions appear, in particular created as RootOfs as the representation of each requires in the loop on line 32. The real algebraic numbers form the static local cell bounds found from the univariate lifting polynomials for c. The real algebraic functions are used to form the local cell descriptions for cells. The real algebraic functions are formed from the relevant projection polynomials that the lifting polynomials come from (via the table T, defining that map).
- We must be careful to take into account any bounds donated from any lifting constraints (Section 3.6).

Algorithm 15 represents propagation of truth values to attempt to deduce QE, removing subtrees from the CAD tree that can be seen as superfluous due to propagation of truth values. It must be called on an evaluated `CADCell` "cell" with child cells, and so always follows usage of Algorithm 14 on "cell". The canonicalization of the algorithm uses the object properties of a `CADCell`. The idea is to iterate across the child cells, evaluating them to attempt to deduce their truth value, or inspecting their truth value if it was already evaluated. Depending on the quantifier (or lack thereof) commensurate with the canonical level of "cell", the determinate truth value of a child cell may be meaningful to the extent the parent can immediately share that truth value. In other cases the same particular truth value amongst all child cells allows the parent to take the same truth value. Whenever a cell receives a determinate truth value, it is added to "leaves" and removed from "cad", because further stack construction is unnecessary. Furthermore, the whole subtree beneath such a cell can be "removed from the subtree" in terms of removal from "cad", if one existed already. The addition of a cell to "leaves" is performed by evalAndSetTruthValue, the function also performing evaluation (Section 3.4).

Some remarks on nuances of this formulation of PRPTV, Algorithm 15:

1. While there is scope for strategy amongst the child cells to evaluate first, the current formulation and implementation includes none, merely iterating across them in order. Some obvious scope for strategy is to evaluate the local sectors first, because evaluation (of a formula to assign a cell's tarski_formula) in terms of a local sample point featuring a rational number is less costly than the evaluation

Algorithm 14 Creation of child cells for a `CADCell`

Input: c, a level $0 \leq k < n$ `CADCell`, cad a container for new unevaluated cells, bases a `projection` object, vars an `Array` of all variables (corresponding to the variable ordering $x_1, \ldots, x_n$), n number of variables for CAD, OpenCAD a boolean flag dictating whether we construct an open CAD i.e. ignore building of sections, constraints and bounds `Array`s created from parsing of lifting constraints, curtainCheck a boolean flag by keyword option, by default *true*

Output: No meaningful return, but $c \mapsto$ children populated with the new child cells of c, all added to cad

1: **procedure** CCHILD(c, cad, bases, vars, OpenCAD, constraints, bounds, curtainCheck)
2: Let α be the full sample point for c
3: Let lvl be $k + 1$
4: **if** curtainCheck **and** $n > 1$ **and** lvl > 1 **and** detectLazardCurtain(c, bases) **then** ▷ Code Fragment 27
5: **ERROR** — Lazard curtain detected — the level $k + 1$ equational constraint chosen as pivot $\prod_{f \in C_{\text{lvl}}} f$ has non zero valuation on c
6: **end if**
7: (B, T) ← univariateBasisAtLazard(bases, c)
8: Let x be vars[lvl] i.e. x_{lvl}
9: **if** constraints and bounds were passed **then**
10: (lb, ub) ← bounds[lvl][1], bounds[lvl][2] ▷ The lower and upper bounds as real numbers formed by lifting constraints at this level — Section 3.6
11: (intervals, polys) ← isolateRootsBasis(B, x, constraints, bounds)
12: **else**
13: (lb, ub) ← $-\infty, \infty$
14: (intervals, polys) ← isolateRootsBasis(B, x)
15: **end if**
16: $n_{\text{roots}} \leftarrow |\text{intervals}|$
17: **if** $n_{\text{roots}} = 0$ **then** ▷ No roots, so only one sector to construct
18: **if** lb $= -\infty$ **then**
19: **if** ub $= \infty$ **then**
20: cell ← `CADCell`(*true*, c, lvl, 1, *true*, $-\infty$, ∞)
21: **else**
22: cell ← `CADCell`(ub $< x$, c, lvl, 1, *true*, $-\infty$, ub)
23: **end if**
24: **elseif** ub $= \infty$ **then**
25: cell ← `CADCell`(lb $< x$, c, lvl, 1, *true*, lb, ∞)
26: **else**
27: cell ← `CADCell`(lb $< x \wedge x <$ ub, c, lvl, 1, *true*, lb, ub)
28: **end if**
29: push(cad, cell)

Algorithm 14 Creation of child cells for a `CADCell`, Part 2

30: **else**
31: (raf, ran) ← two empty `Array`s with n_{roots} elements
32: **for** i **to** n_{roots} **do**
33: $j \leftarrow 1 + \sum_{j=1}^{i-1} \mathbb{1}_{T[\text{polys}[i]]=T[\text{polys}[j]]}$
34: raf$[i] \leftarrow \text{RootOf}(T[\text{polys}[i]], x, \text{index} = \text{real}[j])$
35: ran$[i] \leftarrow \text{RootOf}(\text{polys}[i], x, \text{intervals}[i][1]..\text{intervals}[i][2])$
36: **end for**
37: **if** lb $= -\infty$ **then** ▷ "Left-most" sector
38: push(cad, `CADCell`($x <$ raf[1], c, lvl, 1, *true*, $-\infty$, ran[1]))
39: offset $\leftarrow 0$
40: **elseif** lb $<$ intervals[1][1] **then**
41: push(cad, `CADCell`(lb $< x \wedge x <$ raf[1], c, lvl, 1, *true*, lb, ran[1]))
42: offset $\leftarrow 0$
43: **else**
44: offset $\leftarrow 1$
45: **end if**
46: **if** OpenCAD **then**
47: **for** i **to** $n_{\text{roots}} - 1$ **do**
48: push(cad, `CADCell`(raf$[i] < x \wedge x <$ raf$[i+1]$, c, lvl, $2(i - \text{offset}) + 1$, *true*, ran$[i]$, ran$[i+1]$))
49: **end for**
50: **else**
51: **for** i **to** $n_{\text{roots}} - 1$ **do**
52: push(cad, `CADCell`($x =$ raf$[i]$, c, lvl, $2i$, *true*, ran$[i]$))
53: push(cad, `CADCell`($x <$ raf$[i] \wedge x <$ raf$[i+1]$, c, lvl, $2i+1$, *true*, ran$[i]$, ran$[i+1]$))
54: **end for**
55: **end if**
56: **if** ub $= \infty$ **then** ▷ "Right-most" sector
57: push(cad, `CADCell`(raf$[-1] < x$, c, lvl, $2n_{\text{roots}} + 1$, *true*, ran$[-1]$, ∞))
58: **elseif** ub $>$ intervals$[-1][2]$ **then**
59: push(cad, `CADCell`(raf$[-1] < x \wedge x <$ ub, c, lvl, $2n_{\text{roots}} + 1$, *true*, ran$[-1]$, ub))
60: **end if**
61: **end if**
62: **return**
63: **end procedure**

Algorithm 15 Propagation of truth values in QE by CAD (Partial CAD) to identify and remove CAD subtrees not required for evaluation or stack construction

Input: cell a level $0 \leq k < n$ `CADCell` with child cells, quants, the `Array` of quantifiers for the CAD, m the number of quantifiers, n the total number of variables, cad a container for unevaluated `CADCell`s, leaves a container for meaningful leaf `CADCell`s, problemCells a container for lifting failures

Output: *false* if a meaningful truth value could be deduced, and a CAD subtree could reasonably be removed at this `CADCell` or higher, else *true*

1: **procedure** PRPTV(cell, quants, m, n, cad, leaves, problemCells)
2: Let k = cell $\mapsto$ level
3: **if** $k + 1 < n - m + 1$ **then** ▷ In unquantified space
4: (ortest, andtest) $\leftarrow$ *false*, *true*
5: **for** c **in** cell $\mapsto$ children **do**
6: $t \leftarrow$ evalAndSetTruthValue(c, leaves)
7: ortest $\leftarrow$ ortest **or** t
8: andtest $\leftarrow$ andtest **and** t
9: **if** t = *true* **or** t = *false* **then**
10: Remove c from cad
11: **end if**
12: **end for**
13: **if not** ortest **then**
14: cell $\mapsto$ truth_value $\leftarrow$ *false*
15: Remove all cells from the CAD subtree rooted at cell from cad and problemCells
16: **return** cell $\mapsto$ level = 0 **or** PRPTV(cell $\mapsto$ parent, quants, m, n, cad, leaves, problemCells)
17: **elseif** andtest **then**
18: cell $\mapsto$ truth_value $\leftarrow$ *true*
19: **end if**
20: **elseif** $Q_{k-n+m+1} = \exists$ i.e. quants$[k - n + m + 1] = \exists$ **then**
21: andtest $\leftarrow$ *false*
22: **for** c **in** cell $\mapsto$ children **do**
23: **try**
24: **if** (ortest $\leftarrow$ ortest **or** evalAndSetTruthValue(c, leaves)) **then**
25: cell $\mapsto$ truth_value $\leftarrow$ *true* ▷ Meaningful truth value
26: **if** cell $\mapsto$ level = 0 **or** PRPTV(cell $\mapsto$ parent, quants, m, n, cad, leaves, problemCells) **then**
27: Remove all cells from the CAD subtree rooted at cell from cad and problemCells
28: **end if**
29: **end if**
30: **catch** "Could not evaluate truth value":
31: lastFailure $\leftarrow$ [c,"Could not evaluate truth value"]
32: **end try**
33: **end for**

Algorithm 15 Propagation of truth values, Part 2

```
34:         if lastFailure is a list then
35:             Add lastFailure to problemCells
36:         elseif not ortest then
37:             cell ↦ truth_value ← false
38:             if cell ↦ level = 0 or PRPTV( cell ↦ parent, quants, m, n, cad, leaves,
                    problemCells ) then
39:                 Remove all cells from the CAD subtree rooted at cell from cad and
                        problemCells
40:             end if
41:         end if
42:     else                                   ▷ Q_{k−n+m+1} = ∀ i.e. quants[k − n + m + 1] = ∀
43:         ortest ← false
44:         for c in cell ↦ children do
45:             try
46:                 if not ( andtest ← andtest and evalAndSetTruthValue( c, leaves ) )
                        then
47:                     cell ↦ truth_value ← false                ▷ Meaningful truth value
48:                     if cell ↦ level = 0 or PRPTV( cell ↦ parent, quants, m, n, cad,
                            leaves, problemCells ) then
49:                         Remove the CAD subtree rooted at cell from cad and
                                problemCells
50:                     end if
51:                 end if
52:             catch "Could not evaluate truth value":
53:                 lastFailure ← [c,"Could not evaluate truth value"]
54:             end try
55:         end for
56:         if lastFailure is a list then
57:             Add lastFailure to problemCells
58:         elseif andtest then
59:             cell ↦ truth_value ← true
60:             if cell ↦ level = 0 or PRPTV( cell ↦ parent, quants, m, n, cad, leaves,
                    problemCells ) then
61:                 Remove all cells from the CAD subtree rooted at cell from cad and
                        problemCells
62:             end if
63:         end if
64:     end if
65:     return true
66: end procedure
```

of a sample point featuring a non trivial real algebraic number (i.e. RootOf), that can only arise in the evaluation of local sections. Local sectors can always attribute a rational number as its local sample point. Of course this strategy is motivated in terms of the early termination criteria enabled by propagation of meaningful truth values, so avoiding more costly evaluation would be good news.

2. Lifting failure avoidance (Section 3.7.2) attempts to ignore failures to evaluate child cells, because if we can propagate a truth value from below, such a failure is irrelevant, because we remove the failed cell from the CAD tree in any case.

3. The return value of PRPTV allows us to deduce the largest possible CAD subtree to remove, instead of removing pointlessly attempting to remove every subtree of that subtree on the way up.

Definition 41 (Meaningful Leaf `CADCell`). *A* meaningful leaf *`CADCell` c is a cell holding a meaningful truth value for the quantifier commensurate with its level, i.e.* $Q_{c \mapsto \text{level}}$*, such that the truth value coincides with its held* tarski_formula*, and the truth value propagates upwards towards the root at least once in Algorithm 15.*

Evaluation of `CADCells`

Upon creation of a `CADCell` (for example in CCHILD (Algorithm 14)), the data it holds at that time is relatively rudimentary — essentially the data the constructor is passed. For a cell of positive level, upon creation it holds:

- its level,
- its lower and upper bounds (if it is a local sector), or the real algebraic number it corresponds to (if it is a local section),
- its parent `CADCell`,
- and its local_index.

A cell that holds only this data is *unevaluated*. The act of evaluation of a `CADCell` c is to, in order:

1. Assign its sample_point via the method getSamplePoint(c). Deducing this sample point is trivial when the cell is a local section (the real algebraic number as local bound for the cell is the only candidate for the local sample point). When the cell is a local sector, we can choose the simplest rational number between the real algebraic numbers as local lower and upper bounds for the cell as sample point,

and then, only when doing QE by Partial CAD:

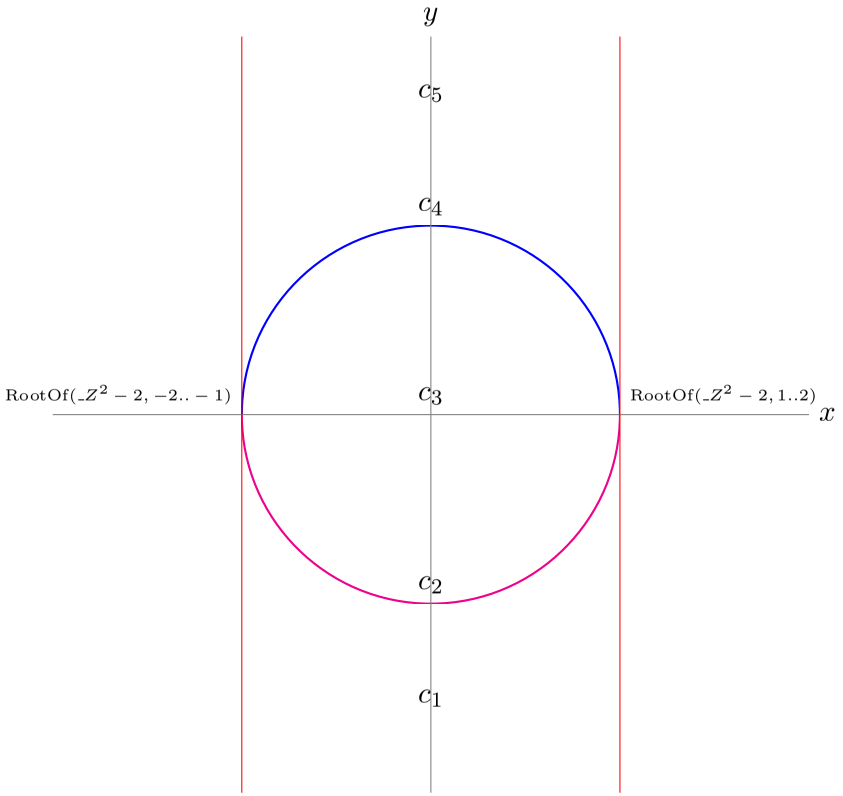
2. Assign its tarski_formula by substitution of its newly found sample_point into the tarski_formula for $c \mapsto$ parent, and additionally its tarski_formula_structural (Tarski formula in structural form) if requested

3. Assign its truth_value. If $c \mapsto$ tarski_formula is equivalent to *true* or *false*, $c \mapsto$ truth_value $\leftarrow c \mapsto$ tarski_formula, else we set $c \mapsto$ truth_value to the Maple value *FAIL*, which types as a boolean value, but means "indeterminate". In the case of a determinate truth value, c is a leaf cell (not necessarily a meaningful one) and is added to the container leaves, else c is amenable to further stack construction, hence added to the container cad.

In the case of Partial CAD, PRPTV instigates this evaluation on all child cells of the last cell to enter CCHILD in turn. In particular evalAndSetTruthValue is the function that instigates all steps 1 through 3 in turn. It returns $c \mapsto$ truth_value for a cell c such that PRPTV can deduce when c is a meaningful leaf. evalAndSetTruthValue is Maple technical due to the substitutions of real algebraic numbers into relations. It takes an argument of leaves such that it adds cells with meaningful truth values to that container. It has the ability to store formulae in structural form for cells. The delay in evaluation of such cells (i.e. the reason why we do not immediately assign these properties as a result of using the constructor) is to avoid the pointless expense of, for example, the algebraic substitution of the new local sample point when we may evaluate a meaningful leaf cell amongst the newly created child cells that allows us to avoid evaluating further child cells. If a cell's truth value has already been set, either by evaluation or by override by propagation of truth values in PRPTV, evalAndSetTruthValue returns that truth value instead ($c \mapsto$ truth_value).

Because the root cell gets assigned Φ as its tarski_formula in QE by Partial CAD (Algorithm 13), any cell of non trivial level stores Φ evaluated at its full sample point as its tarski_formula. However, because it shares all but its last element of its full sample point with the cell above, we need only evaluate the formula of its parent at its local sample point. Hence we reuse as much information from the parent as possible, substituting for one variable per `CADCell` instead of substituting $\mathcal{O}(n)$ values into Φ on every cell evaluation. Hence Φ "filters" down through the CAD tree in order to attempt to deduce meaningful truth values amongst the leaves (or earlier if possible). The tarski_formula property for a `CADCell` represents the equivalent of the unquantified part of input Φ evaluated at the full sample point of that cell, and is guaranteed truth invariant over the whole cell due to sign invariance of the cell on all polynomials from Φ. Hence where tarski_formula is equivalent to *true* or *false*, this allows us to be confident in deducing the truth_value. Once a cell has evaluated and if its truth value is indeterminate, the last property to be assigned at this stage is the children `Array`, initialised and populated by CCHILD.

For full CAD, i.e. CylindricalAlgebraicDecompose, truth values and hence Tarski formulae are superfluous, as the intention is to construct every possible level n cell. Hence steps 2 and 3 are needless. Without purpose to call PRPTV, any unevaluated cell of level $< n$ can enter stack construction via CCHILD, where the generation of its local sample point is required for the full sample point of that cell in order to use lazardEval.



Cell	Visualisation	Description (cell_description)	Local Sample Point (sample_point)
c	Horizontal space between red lines	$\text{RootOf}(x^2 - 2, -2..-1) < x \wedge x < \text{RootOf}(x^2 - 2, 1..2)$	$x = 0$
c_1	Below magenta arc	$y < \text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[1])$	$y = -3$
c_2	Magenta arc	$y = \text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[1])$	$y = \text{RootOf}(\_Z^2 - 2, -2..-1)$
c_3	Between magenta & blue arcs	$\text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[1]) < y < \text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[2])$	$y = 0$
c_4	Blue arcs	$y = \text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[2])$	$y = \text{RootOf}(\_Z^2 - 2, 1..2)$
c_4	Above blue arc	$\text{RootOf}(\_Z^2 + y^2 - 2, \text{index} = \text{real}[2]) < y$	$y = 3$

Figure 3-8: Demonstration of cell parenting and its relation to the cylinder of a cell. We examine a portion of the CAD of $x^2 + y^2 - 2$. $c_1, \ldots, c_5$ are level 2 children of the level 1 cell c. Hence they all share the common element of their full sample point $x = 0$, with 0 the simplest rational in $(\text{RootOf}(x^2 - 2, -2..-1), \text{RootOf}(x^2 - 2, 1..2))$. Each $c_i \subseteq c$, and implicitly shares the local (and full) cell description of c, which is over real algebraic numbers as c is a level 1 cell. All cells share the common ancestor of the root cell, which represents $\mathbb{R}^2$.

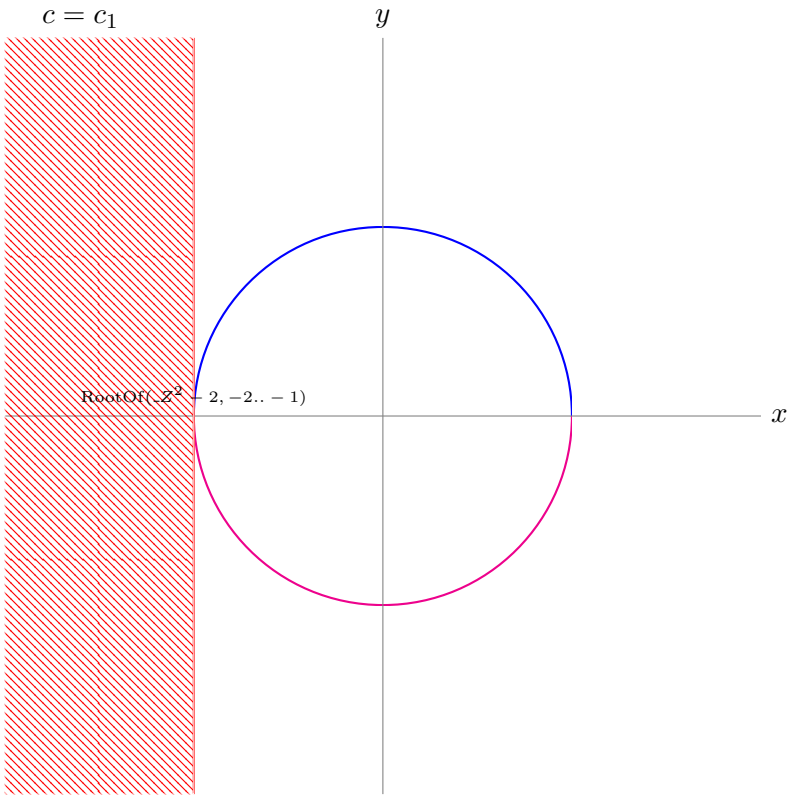


Figure 3-9: Looking at the leftmost part of the CAD from Figure 3-8. There are no roots of $x^2 + y^2 - 2$ in $\mathbb{R}^2$ in the space defined by $x <$ RootOf($\_Z^2 - 2, -2.. - 1$), so when the level 1 cell c (red hatched space, and further north, south, and west) with description $x <$ RootOf($\_Z^2 - 2, -2.. - 1$) enters CCHILD for stack construction, no roots are generated in y, so it generates one child cell c_1 with local description *true*, meaning that c_1 locally corresponds to all values of $y \in \mathbb{R}$. As such the decomposition of c_1 is no finer than c, and their representation of real space is equal. While geometrically they represent the same space, c_1 is a distinct object to c. Their full cell descriptions are both $x <$ RootOf($\_Z^2 - 2, -2.. - 1$).

Cell Descriptions for `CADCells` and QE Output by CAD

Much like a full index or full sample point for a `CADCell`, a full cell description for a cell can be generated by conjunction of its local description with the local descriptions from its ancestral cells. The full description for a cell is a relation (of the form $f \, \rho \, 0$ where f is a real algebraic function and $\rho \in \{<, =\}$), a conjunction of such relations, or *true*. Cell descriptions are important not only cosmetically when inspected by a user (see Section 6.2), but more so as they are pivotal in forming quantifier free output when performing Partial CAD. For QE input $Q_{n-m+1}x_{n-m+1} \dots Q_n x_n \ \Phi(x_1, \dots, x_n)$, the quantifier free equivalent is the truth value of the root cell, unless this truth value is indeterminate (*FAIL*), in which case the quantifier free equivalent is

$$\bigvee_{\substack{c \in \text{leaves} \mid c \mapsto \text{truth\_value} = \mathit{true}, \\ p \mapsto \text{truth\_value} \neq \mathit{false},\ p \text{ on the path from } c \text{ to the root cell}}} \text{GetUnquantifiedDescription}(c) \quad (3.1)$$

where "GetUnquantifiedDescription" refers to the restriction of a full cell description just to the conjunction of those operands of the description not featuring quantified variables. As such (3.1) ends up being in disjunctive normal form. The production of the unquantified part of a cell description is easily realised by modification of Algorithm 16, only gathering local descriptions from the ancestors of a cell in unquantified space. Importantly we discard operands owing to cells in the cylinder of a cell that have a *false* truth value received via propagation of truth values when building (3.1). PRPTV could remove subtrees of cells with a *false* truth value from the container leaves such that we can ignore the conditions past $c \mapsto \text{truth\_value} = \mathit{true}$, but upon iterating over leaves to build the full cell description of each leaf cell we need to traverse towards the root cell anyway in order to construct the full cell description, so we can check truth values of each ancestral cell at the same time and discard those in the cylinder of a cell with a *false* truth value from the disjunction. If (3.1) is an empty disjunction, it is of course equivalent to *false*.

As per Definition 39, the local cell description for a level $i > 0$ `CADCell` is a relation $x_{n-i+1} \, \rho \, f$ where f is a real algebraic function (Definition 31) over $x_{n-i+2}, \dots, x_n$, or a conjunction of two such relations (both strict inequalities). Then, for a level 1 cell, the cell description is in reality over real algebraic numbers due to the lack of dependency on other variables, and CCHILD uses interval indexes in any RootOfs in the cell descriptions created. For a cell of any other positive level, the description may depend on the further variables, so CCHILD stores genuine real algebraic functions where RootOfs are indexed by real indices. The bounds for x_{n-i+1} on this cell depend on $x_{n-i+2}, \dots, x_n$, and so generally we only know the fixed real indices of multivariate polynomials that vary in terms of the values of $x_{n-i+2}, \dots, x_n$.

Cell descriptions are formed by traversal from the cell towards the root cell, so the local descriptions are in successively fewer variables. Forming the conjunction of those local descriptions gathered together is sufficient, but does not necessarily reflect the simplest description given any equalities that can be forward substituted into further elements of the conjunction. For example, the cell description $y = 0 \wedge x < y$ for a CAD

over $[x, y]$ in $\mathbb{R}^2$ can more succinctly be represented by $y = 0 \wedge x < 0$ by substituting the value for y into the real algebraic function as upper bound for x. The level 1 cell on the path to the root cell is that with a univariate cell description, so we need to traverse all the way to the root to gather all elements of the conjunction before performing such substitutions. Code Fragment 16 demonstrates the general methodology for constructing a cell description.

Line 13 is the line corresponding to the check "$p \mapsto$ truth_value $\neq$ *false*, p on the path from c to the root cell" from (3.1). If constructing a cell description in isolation, this check is irrelevant. Else, when constructing the overall disjunction for quantifier free output, we know to discard contribution from this cell to the disjunction to build, as this cell has an ancestor with a *false* truth value.

We substitute as many real algebraic numbers for variables forward through the cell description as many times as the cell c is contiguously a local section (line 28). In fact, as far as we can do this, the cell description owes entirely to exact values, so we replace those elements of the cell description with equations featuring real algebraic numbers, and forward substitute them through the rest of the cell description. Note that there are as many contiguous equations in out as the number of times c is contiguously a local section beginning from x_1. Only local sections contribute a local equation to a cell description (a cell being a local section is equivalent to its cell description being an equation). Once this loop is finished, we should not replace the descriptions owing to local sections with the respective local sample points, because the description may genuinely depend on subsequent variables, but we can still substitute equations forward to simplify the description and eliminate variables (line 35).

Usage of the loop on line 28 means that a cell description such as $x = 1 \wedge y = \text{RootOf}(\_Z^2 - x, \text{index} = \text{real}[1]) \wedge z < y + x$ evaluates as $x = 1 \wedge y = \text{RootOf}(\_Z^2 - 1, -2..-1) \wedge z < \text{RootOf}(\_Z^2 - 1, -2..-1) + 1$. The replacement of equations with local sample points in that loop means that the description in y is $y = \text{RootOf}(\_Z^2 - 1, -2..-1)$ rather than $y = \text{RootOf}(\_Z^2 - 1, \text{index} = \text{real}[1])$, where the former is preferable due to providing more information about the sign and magnitude of the value of y. These values are substituted into the real algebraic function describing values for z.

A cell description such as $x = 1 \wedge y < 1 \wedge z = \text{RootOf}(\_Z^2 - y, \text{index} = \text{real}[2]) \wedge w < z$ evaluates to $x = 1 \wedge y < 1 \wedge z = \text{RootOf}(\_Z^2 - y, \text{index} = \text{real}[2]) \wedge w < \text{RootOf}(\_Z^2 - y, \text{index} = \text{real}[2])$, where one notes the description for z remains as a real algebraic function, because of the break in contiguity in equations at the description for y.

Where a local cell description is *true*, this is equivalent to the cell description being "the whole real line" with respect to that local axis. We can discard such a local description, considering the intention to build a conjunction of the descriptions. If every local description is *true* on the path to the root, then c actually represented $\mathbb{R}^{c \mapsto \text{level}}$, and the extended Tarski formula to represent this is *true*, and this case is identifiable

Algorithm 16 Construction of a cell description

Input: c a `CADCell`
Output: An extended Tarski formula (strictly one of *true*, a relation $f\ \rho\ 0$ where f is a real algebraic function and $\rho \in \{<, =\}$, or a conjunction of such) describing the space in $\mathbb{R}^n$ represented by c

1: **procedure** GetFullDescription(c)
2: **if** $c \mapsto$ level $= 0$ **then**
3: **return** *true*
4: **elseif** $c \mapsto$ level $= 1$ **then**
5: **return** $c \mapsto$ cell_description
6: **end if**
7: cell $\leftarrow c$
8: out $\leftarrow$ an empty `Array`
9: cells $\leftarrow$ an empty `Array` with $c \mapsto$ level elements
10: $i \leftarrow 1$
11: **repeat**
12: cells$[-i$++$] \leftarrow$ cell
13: **if** cell $\mapsto$ truth_value $=$ *false* **then**
14: c is not a meaningful cell for output when building quantifier free output for QE
15: **elseif** cell $\mapsto$ cell_description is a conjunction **then**
16: Add the operands of cell $\mapsto$ cell_description to out ▷ To prevent nesting conjunctions later
17: **elseif** cell $\mapsto$ cell_description is a relation **then**
18: Add cell $\mapsto$ cell_description to out
19: **end if**▷ And omit adding *true* if that's the local description — superfluous as deemed to be element of a conjunction later
20: cell $\leftarrow$ cell $\mapsto$ parent
21: **until** cell $\mapsto$ level $= 0$
22: **if** $|\text{out}| = 0$ **then**
23: **return** *true*
24: **elseif** $|\text{out}| = 1$ **then**
25: **return** out[1]
26: **else**
27: $i \leftarrow 1$
28: **while** $i \leq |\text{out}|$ **and** out$[-i]$ is an equation **do**
29: out$[-i] \leftarrow$ cells$[i] \mapsto$ sample_point▷ Replace with an equality featuring a real algebraic number
30: **for** j **from** $i + 1$ **to** $|\text{out}|$ **do**
31: out$[-j] \leftarrow$ out$[-j]$ substituting out$[-i]$ into out$[-j]$
32: **end for**
33: i++
34: **end while**

Fragment 16 Construction of a cell description, Part 2

```
35:         for j from i to |out| do
36:             if out[−j] is an equation then
37:                 for k from j + 1 to |out| do
38:                     out[−k] ← out[−j] substituting out[−i] into out[−k]
39:                 end for
40:             end if
41:         end for
42:         return ⋀_{i=1,...,|out|} out[−i] ▷ The Array was built backwards in terms of the
                order of the variables for the CAD — this way the conjunction reads
                as triangular in increasing supersets of variables
43:     end if
44: end procedure
```

by line 22. Via the continuing interest in mutable data structures as to avoid garbage collection and not forming pointlessly nested formulae, we append elements of conjunctions from local descriptions to an `Array` and then create the conjunction of all of them.

(3.1) forms an ETF via the real algebraic functions used to describe `CADCell`s. `QuantifierElimination` currently only offers support for real algebraic functions to describe `CADCell`s, hence QE output by CAD is generally an ETF. `QEPCAD B` [8] offers support for QE output via both Extended Tarski Formulae and Tarski Formulae (and other types of formulae), via developments from the PhD thesis of Brown [12]. In `QEPCAD B`, a Tarski formula is always able to be produced in the case when the CAD is "projection definable". Otherwise, `QEPCAD B` may require addition of projection polynomials to achieve this state and be able to produce a Tarski formula to describe output. The equivalent notions such that `QuantifierElimination` is always able to output a Tarski formula are noted as further work. Other CAD implementations produce a TF as output by default (Section 7.5). ETFs can often be quite concise and meaningful, but other times confusing via the density of information contained in the representation.

3.4.1 Real Root Isolation

In order to lift in CAD, we need to perform real root isolation on a lifting polynomial inherited from a level $0 \leq j < n$ `CADCell` to produce the stack over a cell. Generally, such a lifting polynomial may be over algebraic numbers, i.e. Maple RootOfs. In particular, the RootOfs may be nested. Maple's general purpose isolator RootFinding:-Isolate may not accept polynomials where the RootOfs are too nested (in general it requires the input polynomial to be integral). In this case, the polynomial $p(x) \in \mathbb{R}[x_{j+1}]$ is converted to a k-dimensional triangular system to isolate on with respect to, where $k > 1$ is the number of independent algebraic numbers (i.e. RootOfs) contained within p.

The root isolation algorithms used by `QuantifierElimination` are non incremental — other CAD implementations such as [59] may be able to make use of an incremental root isolator where one exists. Incremental root isolation enables an even more incremental approach to CAD than can be achieved here. In particular, while all cells constructed immediately in stack construction by CCHILD here are unevaluated, an incremental approach to root isolation would allow one to avoid constructing all such child cells immediately, with CAD becoming oriented around construction by individual branches in a similar manner to this work's formulation of VTS.

One notes that the RootOfs in a lifting polynomial to isolate are likely as a result of substitution of sample points owing to local sections from `CADCell`s in Lazard evaluation of a projection polynomial, however projection polynomials can contain RootOfs naturally from real algebraic numbers converted from radical numbers from polynomials in input such as Φ. Hence the methodology for real root isolation below is only relevant when not producing an Open CAD, or anything equivalent to a real algebraic number is contained in input for CAD. The representation of irrational roots in `QuantifierElimination` being real algebraic numbers via RootOfs in this format is somewhat developmental & experimental, in order to explore where further support is needed for low level polynomial operations in Maple for polynomials with such coefficients, hence most of the discussion of this subsection.

Algorithm 17 Conversion of a polynomial containing real algebraic numbers (RootOfs) to a triangular system

Input: $p \in \mathbb{R}[x]$
Output: J, a list of polynomials in $k+1$ variables representing p as a triangular system ($J \subset \mathbb{Q}[x, v_1, \ldots, v_k]$), and S, a list of substitutions from RootOfs to new dummy variables

1: **procedure** CONVERTTRIANGULAR(p)
2: $R \leftarrow$ a list of all independent RootOfs in p, sorted in decreasing order of length
3: $k \leftarrow |R|$
4: $S \leftarrow$ a list of substitutions from brand new sequential symbols $v_1, \ldots, v_k$ to the RootOfs, i.e. $S = [v_1 = R[1], \ldots, v_k = R[k]]$
5: $J \leftarrow [b]$ where b is p evaluated at all the substitutions from S
6: **for** i **to** k **do**
7: Append p_i to J where p_i is the polynomial from $R[i]$ evaluated at $[S[i+1], \ldots, S[k]]$
8: **end for**
9: **return** J, S
10: **end procedure**

- Line 4 is achieved in Maple by creating new "local" variables (they can have any names, but the convenient thing is to index them with 1 through k). These never manifest in CAD output — their purpose is merely to be able to represent the polynomials for the algebraic numbers contained within p. These variables are

guaranteed to be distinct from any instances of variables (that may look to have the same name) on the user's side, including the genuine ones in CAD $x_1, \ldots, x_n$.

- R being sorted in decreasing order of length allows us to achieve line 7. "Length" in this context means usage of Maple's "length" command, which actually measures the number of "words" used to represent any expression. This is sufficient to know that for a RootOf $R[i]$ some $0 < i \leq k$, the polynomial for $R[i]$ may contain (via nesting) at most RootOfs from $\{R[i+1], \ldots, R[k]\}$, allowing us to use fewer of the substitutions from S rather than brazenly attempting to substitute all of them at each iteration.

- The generated triangular system for p is guaranteed to have solutions of zero dimension. b (line 5) is (certainly) in $k+1$ variables, and each polynomial from line 7 is in one variable fewer successively, and as such the last added is univariate.

Real algebraic numbers for CAD in `QuantifierElimination` are of the form $\text{RootOf}(p, a..b)$, and as such directly encapsulate the interval for which the real root lies. Standard usage of Maple's root isolator RootFinding:-Isolate on just the triangular system "tri" will find ALL sets of real roots, including those for the polynomials $[J_2, \ldots, J_k]$ generated for the triangular system from the polynomials in the nested algebraic numbers. But in reality many of these root descriptions would be irrelevant and undesirable, because we already know isolating intervals for roots of the polynomials $[J_2, \ldots, J_k]$ (which in reality came from RootOfs). Those isolating intervals can easily be deduced from S in Maple, because each element of S is of the form $\text{RootOf}(J_i, a_i..b_i) = v_i$, so we need look at the second operand of the first operand of each equation of S to deduce the isolating intervals corresponding to each polynomials. These intervals $(a_2 .. b_2), \ldots, (a_k .. b_k)$ form a hyperrectangle via their product $(a_2, b_2) \times \cdots \times (a_k, b_k)$ with respect to the variables $J_2, \ldots, J_k$. Hence, instead of computing all root descriptions for the triangular system and discarding those not consistent with the intervals deduced from S, it is instead desirable to provide those intervals in order to only produce root descriptions within the hyperrectangle formed by them in $\mathbb{R}^{k-1}$. In this case, we have $k-1$ existing intervals for $[J_2, \ldots, J_k]$ amongst a total of k polynomials. In providing these intervals $(a_2 .. b_2), \ldots, (a_k .. b_k)$, we can remove some inefficiency from the root isolation process, and remove the need to manually discard root descriptions out of root isolation outside the box formed by $(a_2 .. b_2), \ldots, (a_k .. b_k)$. Usage of the isolation routine in this way could also refine $(a_2 .. b_2), \ldots, (a_k .. b_k)$ to finer intervals, which we should retain if possible. The passing of such intervals as an argument to restrict the output of real root isolation is a feature in development for the low level root isolation routines used by `QuantifierElimination` in Maple, in part informed and requested as a result of usage of real algebraic numbers in this specific representation.

Root Refinement

Where the isolating intervals for two real roots (from two different lifting polynomials) are not disjoint, they must be made disjoint such that construction of cells (especially

sectors) is well defined. As an extreme example, for the polynomials $x^2 - 2$ and $x^2 - 3$, $(1\,..\,2)$ is an isolating interval for the second (i.e. positive) real root of both. Such isolating intervals could only be obtained at infeasibly low precision in practice. The precision of RootFinding:-Isolate in Maple is controlled by the keyword option `'digits'`, which defaults to the global variable `Digits` if not provided. In any case, where we find an isolating interval that has intersection with an existing interval amongst those from the current polynomial basis to isolate roots for, we must either deduce that the root is the same, or make the isolating intervals disjoint. In the case of this example, the roots happen to be distinct, and it would be sufficient to refine the intervals to $(\frac{11585}{8192}, \frac{2897}{2048})$ and $(\frac{113511}{65536}, \frac{56757}{32768})$ respectively (which again, would be values obtained at bizarrely low precision). refineIsolatingIntervals is the procedure handling this process, which is called by isolateRootsBasis (within CCHILD) and incrementalCADMerge.

The situation of root refinement only arises because we perform root isolation iteratively on factors of lifting polynomials in stack construction. We produce a fully factored irreducible basis for the lifting polynomial obtained by Lazard evaluation, with the decision to produce something fully factored being to assist root isolation in the first place (Remark 34). Usage of RootFinding:-Isolate or an equivalent low level isolator has that the root descriptions returned for one particular polynomial are guaranteed to be disjoint, but we do this iteratively amongst several factors of the lifting polynomial. This is to avoid the needless complexity of isolating roots for the product of all of them, which would provide us with certainly disjoint intervals, at excessive cost, hence the methodology of isolateRootsBasis or incrementalCADMerge to iteratively merge in root descriptions making them disjoint only where appropriate. We go to some length to ensure that the basis creation stores the factors of lifting polynomials canonically (Algorithm 12), but cannot entirely guarantee it due to the usage of real algebraic numbers, which fail to be canonical due to representations using differing valid isolating intervals. Luckily, Maple can deduce if two root descriptions owing to differently represented polynomials are equal by forming the RootOfs about each and using Maple's general purpose `is` — (`is`(RootOf$(p_1, a_1..b_1)$ = RootOf$(p_2, a_2..b_2)$)). This is the first step of the function refineIsolatingIntervals, where the intention is to produce an error that the roots are the same if this check for equality returns *true*. Catching this error notifies isolateRootsBasis or incrementalCADMerge to discard the incoming root description to merge in. Otherwise, the root descriptions should genuinely be different, and we enter root refinement on both descriptions to attempt to make them disjoint up to a certain precision. If this fails, perhaps due to exceeding a precision threshold, the associated stack construction fails via an error (that could be caught via general avoidance of lifting failures in Partial CAD — Code Fragment 28 and also broadly Section 3.7.2), else we should return the refined RootOf expressions in order to appropriately store and potentially make use of this information later (for example, doing so may avoid the necessity to enter root refinement again).

Conversion of a polynomial with coefficients as real algebraic numbers to a triangular system enables real root isolation of such a polynomial under this representation. The same is true of of refineIsolatingIntervals — where either of the polyno-

mials donating isolating intervals feature irrationals represented by RootOfs, we can convert to a triangular system (for either or both respectively) to provide amenable input for the root refinement methods. Let the two intersecting isolating intervals be (a_f, b_f) and (a_g, b_g) for the polynomials f and g respectively. Let f and g form the triangular systems $[f, f_1, \ldots, f_k]$ and $[g, g_1, \ldots, g_m]$ respectively, $k, m \geq 0$ with $[(a_f, b_f), (a_{f_1}, b_{f_1}), \ldots, (a_{f_k}, b_{f_k})]$ and $[(a_g, b_g), (a_{g_1}, b_{g_1}), \ldots, (a_{g_k}, b_{g_k})]$ the respective intervals representing isolating intervals for roots of the respective polynomials from the system. Hence there are k and m irrational numbers represented by interval indexed RootOfs in f and g respectively. Here, in contrast to the case for pure root isolation, the number of intervals corresponds exactly with the number of polynomials for the system ($k+1$ and $m+1$ respectively). In the case for root isolation, we would have k or m intervals for $k+1$ or $m+1$ polynomials. Our main intention is to refine (a_f, b_f) and (a_g, b_g), but the routine for root refinement may actually also refine $[(a_{f_1}, b_{f_1}), \ldots, (a_{f_k}, b_{f_k})]$ and $[(a_{g_1}, b_{g_1}), \ldots, (a_{g_k}, b_{g_k})]$. We should retain this information about the refinement of these intervals if possible. The associated root descriptions for f and g may be stored as cell bounds for local sectors, or the exact cell bound for a local section, and we consider that such bounds may be examined in CAD incrementality (incrementalCADMerge, Algorithm 51) under further refinement. Hence, f and g can (and should) use the finest possible intervals gleaned from refinement on $[(a_f, b_f), (a_{f_1}, b_{f_1}), \ldots, (a_{f_k}, b_{f_k})]$ and $[(a_g, b_g), (a_{g_1}, b_{g_1}), \ldots, (a_{g_k}, b_{g_k})]$, at the very least to cater towards further operations in the context of incrementality.

As usual, further Maple development will optimise the refinement process on polynomials in this representation and maximise the chance of success to mitigate this as a lifting failure (Section 3.7.2). In its current state, the function refineIsolatingIntervals only refines the intervals (a_f, b_f) and (a_g, b_g), using Maple technical low level operations, but development will make the retention of the refinement of the other isolating intervals for the real algebraic numbers coefficients of f and g relevant.

Having discussed the technicalities behind real root isolation on polynomials over real algebraic numbers in `QuantifierElimination`'s CAD, we can present the algorithm to generate and merge in the root descriptions gained iteratively from a univariate polynomial basis such that we can produce an ordered structure of intervals, together with a structure of the univariate polynomials they arose from. We assume isolateRootsOf returns an ordered list of isolating intervals for b in x obliging the lifting constraints via constraints and bounds. The rest of the function mostly reduces to usage of merging in root descriptions, checking for intersection with the closest interval. Where there is an intersection, there is an attempt to refine the (in practice, inertised) RootOfs via refineIsolatingIntervals. As per the previous discussion we should retain *all* information obtained from refinement. If refineIsolatingIntervals finds the roots to not be disjoint at a maximum precision, we discard the incoming isolating interval [lb, ub], and if it fails for any other reason this is caught by general lifting failure avoidance (Section 3.7.2). Algorithm 18 represents generation of root descriptions to accommodate non incremental CAD, i.e. stack construction proceeds immediately from such root descriptions where no stack existed previously below a cell.

Algorithm 18 Root isolation of elements of a univariate polynomial basis via merging of root descriptions

Input: B a basis of polynomials in x only, x a variable, constraints and bounds `Array`s created from parsing of lifting constraints
Output: intervals an ordered `Array` of isolating intervals for roots of polynomials in B, polys an `Array` of the corresponding polynomials each isolating interval in intervals came from

```
1: procedure isolateRootsBasis( B, x, constraints, bounds )
2:     intervals ← an empty Array
3:     polys ← an empty Array
4:     rootCount ← 0
5:     for b in B do
6:         ( i, b′ ) ← 1, b
7:         iso ← isolateRootsOf( b, x, constraints, bounds )
8:         for [lb, ub] in iso do
9:             dontAdd ← false
10:            repeat
11:                while i ≤ rootCount and lb > intervals[i][2] do
12:                    i++
13:                end while
14:                if i ≤ rootCount then
15:                    ( pastLB, pastUB ) ← intervals[i][1], intervals[i][2]
16:                    if pastLB ≤ lb then          ▷ Some sort of intersection between
                           isolating intervals
17:                        pastRtf ← RootOf( polys[i], pastLB..pastUB )
18:                        newRtf ← RootOf( b, pastLB..pastUB )
19:                        if pastUB < ub then
20:                            try
21:                                ( pastRtf, newRtf ) ← refineIsolatingIntervals(
                                       pastRtf, newRtf )
22:                                b′ ← the polynomial from newRtf
23:                                ( lb, ub ) ← the interval from newRtf
24:                                polys[i] ← the polynomial from pastRtf
25:                                intervals[i] ← the interval from pastRtf, as a list
26:                                i++          ▷ If the above succeeded, lb > intervals[i][2]
27:                            catch "Roots the same":
28:                                dontAdd ← true
29:                            end try
```

Algorithm 18 Isolation of a univariate polynomial basis via merging of root descriptions, Part 2

```
30:                     else
31:                         try
32:                             ( pastRtf, newRtf ) ← refineIsolatingIntervals(
                                    pastRtf, newRtf )
33:                             b′ ← the polynomial from newRtf
34:                             ( lb, ub ) ← the interval from newRtf
35:                             polys[i] ← the polynomial from pastRtf
36:                             intervals[i] ← the interval from pastRtf, as a list
37:                         catch "Roots the same":
38:                             dontAdd ← true
39:                         end try
40:                     end if
41:                 elseif pastLB ≤ ub then          ▷ Opposite type of intersection
42:                     try
43:                         ( newRtf, pastRtf ) ← refineIsolatingIntervals( newRtf,
                                pastRtf )
44:                         b′ ← the polynomial from newRtf
45:                         ( lb, ub ) ← the interval from newRtf
46:                         polys[i] ← the polynomial from pastRtf
47:                         intervals[i] ← the interval from pastRtf, as a list
48:                     catch "Roots the same":
49:                         dontAdd ←true
50:                     end try
51:                 end if
52:             end if
53:         until dontAdd or ( i = 1 or lb > intervals[i − 1][2] ) and (
                        i > rootCount or ub < intervals[i][1] )
54:         if not dontAdd then
55:             Insert [lb, ub] at position i in intervals    ▷ Merge in information for
                    root description
56:             Insert b′ at position i in polys
57:             rootCount++
58:         end if
59:     end for
60: end for
61: return intervals, polys
62: end procedure
```

3.4.2 Delayed Evaluation of Substitutions in CAD

Similar for the case for VTS described in Section 2.4.1, CAD supports delayed substitution of sample points into non atomic Tarski formulae. To notify that this is a delayed CAD substitution, the implementation of Partial CAD uses the Maple construct

$$\texttt{`:-inertCADSub`}(\Psi, x = \alpha)$$

which appropriately types as an `RTFArray`. In other words this types as a real Tarski formula expression acceptable to be held by a `CADCell` in CAD. In the same way as VTS, the usage of `` ` ` `` and `:-` is to ensure `inertCADSub` remains unevaluated and is not interpreted in terms of any potential assignments to the global instance of `inertCADSub` by the user or other code.

To make clear when this is relevant, substitutions are always of one algebraic number for one variable into the real Tarski formula as `RTFArray` held by one `CADCell`. If c is a level $i \geq 0$ `CADCell`, then to "evaluate" the truth value of a child cell c_1, we must substitute the local sample point of c_1 into $c \mapsto$ tarski_formula (that is the real Tarski formula as `RTFArray` held by c) for x_{i+1}. This local sample point is an algebraic number. This evaluation occurs in usage of PRPTV, and hence is only relevant in QE by PartialCylindricalAlgebraicDecompose. Hence we receive the tarski_formula for c_1, and may be able to deduce the truth value of c_1 on the unquantified part of the input formula Φ for QE (as is one of the aims of PRPTV).

The delayed evaluation of substitutions in CAD may avoid substitution of algebraic numbers for one variable. Where the algebraic numbers are genuine irrational ones (i.e. the associated univariate polynomial is not linear — a Maple RootOf), to attempt to deduce if a relation $f \, \rho \, 0$ has a determinate truth value, `QuantifierElimination` (using evaluateTFArrayAtSP) must:

- Use `evala( Normal( ... ) )` to produce the normal form for $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)]$. Note that the documentation of `evala(Normal(...))` states that "the result [of `evala(Normal(...))`] is zero only if the input is mathematically equal to zero" (hence the normal form in the context of Definition 20). `evala( Normal( ... ) )` must compute over algebraic field extensions of all the RootOfs contained in $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)]$ (there may be further ones in $f \, \rho \, 0$ via any previous substitutions).
- If $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)]$ is free of variables (from $x_1, \ldots, x_n$), $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)] \not\equiv 0$, $\rho \in \{<, \leq\}$, and $f[x \,/\, \text{RootOf}(p, a..b)]$ is not a monomial of RootOfs, then use Maple's shake to deduce the sign of $f[x \,/\, \text{RootOf}(p, a..b)]$ and hence the truth of $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)]$. If $\rho \in \{=, \neq\}$ and $(f \, \rho \, 0)[x \,/\, \text{RootOf}(p, a..b)]$ is free of RootOfs and any variables then the truth value follows trivially, because of the normalisation by `evala( Normal( ... ) )`.

Both of the elements above attribute a non trivial cost, but less obvious is that usage of shake on highly nested RootOfs may fail (by production of an error from shake).

Altogether this enables delayed evaluation of substitutions in CAD.

In contrast to VTS, substitution in CAD requires Maple to normalise expressions featuring algebraic numbers, and by the mechanism of CAD these RootOfs may be nested. Meanwhile in VTS, we may avoid virtual substitutions from multivariate polynomials into other multivariate polynomials, which generally reduces to pseudoremainders, but via formulae schemes, may also be a recursive process on virtual substitutions of polynomials of descending degree.

3.5 Open CAD

Definition 42 (Open CAD). *An* Open CAD *(OpenCAD) is one that only lifts sectors, i.e. cells of full dimension with fully odd full cell indices.*

Open CAD is a feature for CAD relevant to lifting that ignores creation of sections, i.e. cells with an even local index with a local sample point owing exactly to the real root of some lifting polynomial. It is so named because each resulting CAD cell formed owes entirely to "open" space similar to a product of open intervals, and as such each constructed CAD cell is always of full dimension. Open CAD has been realised before in the work of Sub-CADs [29], which allow for relinquishment of certain types of cell (in this case, sections). There, the discrimination is specifically on cell dimension, which for Open CAD corresponds to full dimension, i.e. dimension n. Sub-CADs are therefore more general than Open CADs.

This is useful when substitution of exact real roots from lifting polynomials can be deduced to be superfluous or undesirable, such as in applications related to motion planning [29]. The former case of superfluity of sections corresponds to the case of an input formula with only strong relations (Algorithm 19). `QuantifierElimination` features OpenCAD functionality for the top level functions CylindricalAlgebraicDecompose and PartialCylindricalAlgebraicDecompose. OpenCAD can also be used in the evolutionary routines InsertFormula and DeleteFormula when acting upon a `CADData` object (Algorithms 54 and 56). The differing methodology of Open CAD in terms of ignoring construction of sections can most easily be recognized as part of Algorithm 14 (CCHILD).

Construction of Open CADs in `QuantifierElimination` is controlled by the keyword option `'OpenCAD'`. Passing of the option `'OpenCAD'` = *true* forces construction of an Open CAD under any circumstances when passed to CylindricalAlgebraicDecompose, considering the intention is to build a full CAD with all leaf cells regardless of truth values. The CAD routines in the context of QE (e.g. PartialCylindricalAlgebraicDecompose) will oblige this flag only when Algorithm 19 returns *true* on the current input formula to perform CAD on, to ensure that the CAD built is actually sufficient to deduce QE (else the output quantifier free answer may be incorrect). In total, Open CAD is a lifting optimisation, however given its "destructive" properties, it is not used without direct specification by the user by keyword option, in case the user wished to inspect every usual leaf cell from output.

Algorithm 19 Algorithm to deduce if a formula has only strong relations (hence Open CAD may be desirable)

Input: Φ, an unquantified prenex (real) Tarski formula
Output: *true*, if Φ contains only strong relations, else *false*

```
1: procedure hasAllStrongRelations( Φ )
2:     if Φ = (f ρ 0) then                          ▷ f ∈ ℝ[x_1, ..., x_n], ρ ∈ {≤, =, <, ≠}
3:         if ρ ∈ {≤, =} then
4:             return true
5:         else
6:             return false
7:         end if
8:     else
9:         for i to the number of operands of Φ do
10:            if not hasAllStrongRelations( Ψ ) where Ψ is the ith operand of Φ
                   then
11:                return false
12:            end if
13:        end for
14:        return true
15:    end if
16: end procedure
```

Apart from the production of fewer cells in the case of Open CAD, the avoidance of usage of exact real roots of lifting polynomials means that we only ever (Lazard) evaluate polynomials or substitute values into formulae via linear equations featuring rational numbers, and never irrationals. In `QuantifierElimination`, irrational numbers are represented by RootOfs, but the aforementioned operations are always clearly less costly on rational numbers.

Tables 3.1 and 3.2 represent data about the number of cells traversed and lifted with full CAD via CylindricalAlgebraicDecompose for the two examples from the `QEExamples` table for this project that are sensible with usage of Open CAD, i.e. the only two formulae featuring only strong relations for which Algorithm 19 would return *true*. All other keyword options used are those as default in `QuantifierElimination`. Because of the presence of CPU time, one notes the examples were performed on a computer running Maple 2020.1 with 16.0 GB of RAM, and a 3.70 GHz Intel i5-9600k processor. The projection sets are always the same as without usage of Open CAD, given that Open CAD acts only within the context of lifting. The same data about number of cells can be inspected by specifying `infolevel[ PartialCylindricalAlgebraicDecompose ]` $:= 10$ before running the same experiments.

Figure 6-1 shows an example of a call to PartialCylindricalAlgebraicDecompose in Maple with passing of `'OpenCAD'` = *true*, where the suggestion to use Open CAD comes from the `QuantifierTools` function SuggestCADOptions (Chapter 6).

OpenCAD	# cells traversed	# leaf cells	CPU Time (s)
true	6	4	0.047
false	10	7	0.062

Table 3.1: Comparison of cells in lifting by CylindricalAlgebraicDecompose (full CAD) & CPU time used with OpenCAD on and off for 'Piano Movers Problem (Yang, Zheng)' from the `QEExamples` table.

OpenCAD	# cells traversed	# leaf cells	CPU Time (s)
true	86	52	0.250
false	317	223	0.422

Table 3.2: Comparison of cells in lifting by CylindricalAlgebraicDecompose (full CAD) & CPU time used with OpenCAD on and off for 'Ball and Circular Cylinder' from the `QEExamples` table.

3.6 Lifting Constraints

`QuantifierElimination` also allows for the passing of "lifting constraints" to the CAD routines CylindricalAlgebraicDecompose and PartialCylindricalAlgebraicDecompose via the keyword option `'LiftingConstraints'` for both functions.

Definition 43 (Lifting Constraint). *A* Lifting Constraint *(LC) is a relation* $f \rho\ 0$ *such that* $\rho \in \{<, \leq\}$, $f \in \mathbb{R}[x_i]$ *for some* $i \in \{1, \ldots, n\}$, *and* $\deg(f) = 1$. *As such, any set of lifting constraints forms a hyperrectangle or "box", i.e. a product of intervals* $[l_1, u_1] \times \cdots \times [l_n, u_n]$ *in* $\mathbb{R}^n$, *where* $l_i \in \mathbb{R} \cup \{-\infty\}$ *and* $u_i \in \mathbb{R} \cup \{\infty\}$ $\forall i = 1, \ldots, n$.

Less formally, lifting constraints are linear relations in one variable each which define a hyperrectangle in real space for CAD to lift in. Categories of QE examples where lifting constraints are relevant are those from mathematical biology or motion planning, where frequently "negative" real space is often superfluous. In a similar manner to equational constraints (Section 3.7), lifting constraints must appear at the top level of an (at most) existentially quantified conjunction to successfully constrain the formula such that lifting in a hyperrectangle is well intentioned. We highlight the differences between including $p(x_i)\ \rho\ 0$ as a lifting constraint and including $p(x_i)\ \rho\ 0$ in the top level conjunction for a formula, for some $1 \leq i \leq n$.

- $p(x_i)\ \rho\ 0$ is included as a lifting constraint:
  - Roots of lifting polynomials in x that do not satisfy $p(x_i)\ \rho\ 0$ are discarded, or ideally not computed at all, if the root isolator supports such restrictions. Hence lifted cells are guaranteed to satisfy $p(x_i)\ \rho\ 0$ with respect to their sample point in x.
  - $p(x_i)$ is linear in x_i, so projection operations on it are trivial. If it were to appear somewhere in projection, it would be in the projection bases for x_i, but it is not included on account of $p(x_i)\ \rho\ 0$ (but may appear if projection

by coincidence via other polynomials from input that were not designated as lifting constraints)

- $p(x_i)\,\rho\,0$ is included in the top level of the (existentially quantified) conjunction Φ to process:
  - $p(x_i)$ certainly appears in the projection bases for x_i (as an inequality). $p(x_i)$ appears as a lifting polynomial as many times as there are unevaluated cells in x_{i-1}, or just once if $i = 1$. $p(x_i)$ is linear, hence only has one root (henceforth $\text{RootOf}(p(x_i), a..b)$, some $a \leq b$ — p can have irrational coefficients), and that root may appear as bounds for cells in x_i. Hence we build geometry such that $x_i < \text{RootOf}(p(x_i), a..b)$, $x_i = \text{RootOf}(p(x_i), a..b)$ (if we are not building an Open CAD), and $x_i > \text{RootOf}(p(x_i), a..b)$. Because p is linear and $\rho \in \{<, \leq\}$, $p(x_i)\,\rho\,0$ will be *false* on at least one of these cells, hence at least one of these cells holds the truth value *false*.

Usage of lifting constraints should hence be viewed as an optimisation when one knows the simple facts about the geometry of the system, or where one wants the solutions to lie. However, usage will omit production of cells that may be desirable for examination, if one need know about all the geometry and sign conditions on such. As a similarly "destructive" lifting optimisation as Open CAD, it is generally not used without direct specification by the user.

Lifting constraints are rarely implicit within a formula, at least in a non trivial sense in the manner of implicit equational constraints (Section 3.7). For example, the clause $x > 1 \vee x > 2$ is equivalent to $x > 1$, and so within a conjunction really represents one lifting constraint. The need for such constraints to be linear in one variable limits their ability to appear implicitly. One notes that a pair of lifting constraints that are immediately unsatisfiable, such as $x > 1 \wedge x < 0$ imply degenerate geometry if parsed as lifting constraints, and `QuantifierElimination` CAD functions will produce an error in this case upon parsing the lifting constraints (line 24 of Algorithm 20). In reality an inconsistent set of constraints such as $x > 1 \wedge x < 0 \equiv \mathit{false}$, and so the related formula is certainly *false* because $x > 1 \wedge x < 0$ must exist as operands of a top level conjunction, but `QuantifierElimination` cannot proceed with lifting with the bounds implied by these lifting constraints because the cell bounds will be unintelligible (e.g. viewing Algorithm 14). An error produced in this case makes it clear to the user that these constraints are indeed unsatisfiable, so the formula is equivalent to *false*. A disjunction such as $x > 1 \vee x < -1$ does *not* fit in the framework of lifting constraints, because it is inconsistent with Definition 43, which requires such constraints to form the interior of a single box, and not the union of such boxes.

A set of lifting constraints $\{x_1\,\rho_{1,1}\,\alpha_{1,1}, x_1\,\rho_{1,2}\,\alpha_{1,2}, \ldots, x_n\,\rho_{n,1}\,\alpha_{n,1}, x_n\,\rho_{n,2}\,\alpha_{n,2}\}$ is equivalent to the conjunction

$$\bigwedge_{i=1}^{n} (x_i\,\rho_{i,1}\,\alpha_{i,1}) \wedge (x_i\,\rho_{i,2}\,\alpha_{i,2})$$

Lifting Constraints Passed	# cells traversed	# leaf cells	CPU Time (s)
Via formula	11949	7129	33.05
Via option	6732	3662	35.88

Table 3.3: Comparison of cells in lifting by CylindricalAlgebraicDecompose with lifting constraints passed by option and as part of the formula for 'Ellipse A' from the QEExamples database.

Lifting Constraints Passed	# cells traversed	# leaf cells	CPU Time (s)
Via formula	?	?	?
Via option	627	442	2.39

Table 3.4: Comparison of cells in lifting by CylindricalAlgebraicDecompose with lifting constraints passed by option and as part of the formula for 'Collision' from the QEExamples database. '?' notifies that QE by CAD (PartialCylindricalAlgebraicDecompose) could not be completed within a standard timeout of 750 seconds.

where $\alpha_{i,j} \in \mathbb{R} \cup \{-\infty, \infty\}$ and $\rho_{i,j} \in \{<, \leq\}$, $\forall i \in \{1, \ldots, n\}$, $\forall j = 1, 2$. Any lifting constraints with ∞ or $-\infty$ can essentially be ignored, always being trivially satisfiable.

Tables 3.3 and 3.4 are tables of data comparing usage of lifting constraints via keyword option with their standard use being passed in the formula, when producing a full CAD via CylindricalAlgebraicDecompose. All other keyword options used are those as default in `QuantifierElimination`. Because of the presence of CPU time, one notes the examples were performed on a computer running Maple 2020.1 with 16.0 GB of RAM, and a 3.70 GHz Intel i5-9600k processor. The projection sets are always the same, given that lifting constraints act only within the context of lifting. The same data about number of cells can be inspected by specifying `infolevel[ PartialCylindricalAlgebraicDecompose ]` := 10 before running the same experiments.

The symbol `'positive'` can also be passed for the `'LiftingConstraints'` option such that we only build the CAD in positive real space, i.e. $\mathbb{R}^n_+$ or $(0, \infty) \times \cdots \times (0, \infty)$, being a shortcut for passing of the set $\{x_1 > 0, \ldots, x_n > 0\}$ as lifting constraints.

Usage of Algorithm 20 parses the given lifting constraints to check their consistency, and find the smallest possible intervals to describe the maximal bounds for cells to build per each variable (i.e. the lower and upper bound for left-most and right-most sectors respectively). It returns two `Array`s, bounds and constraints, which can be used by the two main lifting routines CCHILD (Algorithm 14) and incrementalCADMerge (Algorithm 51). The former does standard lifting while the latter does incremental lifting amongst existing cells. More than two lifting constraints per variable is definitely superfluous, possibly even inconsistent (line 24), but it's up to the user to formulate them well (or formulate the example well), and we always end up iterating over an arbitrary number of them as a result (usually up to two). If the lifting constraints form degenerate geometry, i.e. are inconsistent, Algorithm 20 generates an error not

Algorithm 20 Parsing of Lifting Constraints in CAD

Input: vars, an `Array` of all variables for the CAD (corresponding to the ordering $x_1, \ldots, x_n$), n, the positive integer representing the number of variables for the CAD, lc, a set of all lifting constraints (Definition 43) given from top level CAD input, or the symbol `'positive'`

Output: bounds, an `Array` of lists of two real numbers (or $\pm\infty$) representing lower and upper bounds per canonical CAD level, and constraints, an `Array` of sets of lifting constraints relevant to each canonical CAD level

1: **procedure** MANAGELIFTINGCONSTRAINTS(vars, lc)
2: bounds $\leftarrow$ an empty `Array` with n elements
3: constraints $\leftarrow$ an empty `Array` with n elements
4: **if** lc is the symbol `'positive'` **then**
5: **for** i **to** n **do**
6: bounds$[i] \leftarrow [0, \infty]$
7: constraints$[i] \leftarrow \{0 < x_i\}$
8: **end for**
9: **else**
10: **for** i **to** n **do**
11: constraints$[i] \leftarrow \{f \; \rho \; 0 \mid f \; \rho \; 0 \in \text{lc}, \; f \in \mathbb{R}[\text{vars}[i]]\}$ ▷ vars$[i] = x_i$
12: (lb, ub) $\leftarrow \infty, -\infty$
13: **for** $f \; \rho \; 0$ **in** constraints$[i]$ **do**
14: Let $f = ax_i + b$ ▷ f certainly linear in x_i as a lifting constraint
15: **if** $a < 0$ **then**
16: **if** $\frac{b}{|a|} <$ lb **then**
17: lb $\leftarrow \frac{b}{|a|}$
18: **end if**
19: **elseif** $-\frac{b}{|a|} >$ ub **then**
20: ub $\leftarrow -\frac{b}{|a|}$
21: **end if**
22: **end for**
23: **if** ub $\leq$ lb **then**
24: **ERROR** — lifting constraints are inconsistent and give degenerate geometry to build in, hence formula equivalent to *false*
25: **end if**
26: bounds$[i] \leftarrow$ [lb, ub]
27: **end for**
28: **end if**
29: **return** bounds, constraints
30: **end procedure**

to be caught, notifying the user that the formula Φ is trivially equivalent to *false*. If this occurs in the context of QE, this implicitly provides QE output, albeit not in the conventional sense of a parsable formula.

The total semantics in terms of arguments for the root isolation wrapper isolateRootsOf used by CAD in `QuantifierElimination` are that it takes a univariate polynomial $b \in \mathbb{R}[x]$, the variable x, a set of lifting constraints c in x from the `Array` constraints, and a lower and upper bound as a list $[\text{lb}, \text{ub}]$, again corresponding to the variable x, and from the `Array` bounds. isolateRootsOf directly obliges the lifting constraints by returning an ordered list of isolating intervals for roots of b that all satisfy the lifting constraints c .

One notes that, for example, the functionality of RootFinding:-Isolate in Maple provided by the keyword option `'constraints'` evaluates a list of polynomials at each of the root descriptions found in isolation. Given the precision involved in doing so, this actually provides intervals about the evaluation those polynomials at each root. Therefore for a set of lifting constraints constraints $= \{f_1 \, \rho \, 0, \ldots, f_k \, \rho \, 0\}$, isolateRootsOf passes the list $[f_1, \ldots, f_k]$ as `'constraints'` to a low level isolation procedure, and then evaluates the associated relations, discarding each root unless the evaluation of all these relations are *true*. Passing of constraints to the isolation procedures used is generally only feasible where the lifting constraints are integral — those that contain real algebraic numbers must be evaluated at each root manually, which is not as reliable. If the evaluation of an irrational lifting constraint fails, this failure can be ignored if evaluation of another lifting constraint is *false*, in which case the failure to evaluate another constraint at the same root description is immaterial. Hence isolateRootsOf passes the integral polynomials from amongst $[f_1, \ldots, f_k]$ as `'constraints'` providing reliable evaluation of the associated constraints, and any irrational polynomials are evaluated manually by usage of Maple's shake about the root description to try to deduce the sign of the constraint. When the lifting polynomial to isolate is completely integral, the low level isolation routine to use can accept a real interval arising from the pair of bounds in x produced from parseLiftingContraints, automatically excluding production of root descriptions outside this interval, however that routine actually assumes it forms a closed interval. Therefore we need to check the satisfiability of all the strict inequalities amongst our lifting constraints per returned isolating interval (these are the only constraints that could make that interval open at either end instead). However, when the lifting polynomial is integral, the univariate isolation procedure *can* actually evaluate even irrational lifting constraints. This is why not only an `Array` of bounds is returned by parsing of lifting constraints, but an `Array` of the sets of lifting constraints per variable, as it is rarely sufficient to just pass an interval to restrict root isolation. In total, the only potentially unreliable case for evaluation of lifting constraints is that of a lifting polynomial with irrational coefficients and irrational lifting constraints, and the only (rare) cause for failure, but this only has us erroring out of CAD completely if the lifting recovery methods discussed via and after Section 3.7.2 are either inapplicable (full CAD) or fail to let us deduce a quantifier free equivalent despite this failure to evaluate constraints and hence produce isolating intervals about roots in order to

construct the stack on a cell (Partial CAD). The discussion of developments to low level real root isolation routines to better accommodate the case for polynomials over real algebraic numbers (Section 3.4.1) applies similarly here, such that evaluation of `'constraints'` continues to be supported to at least the same extent as the discussion here.

The functionality of lifting constraints to restrict root isolation finds itself useful in a further context via Algorithms 31 and 34 from Section 3.7.2. Here, we narrow down bounds for root isolation from any lifting constraints passed via the local sample points of neighbouring cells of cells with curtains. As the bounds and associated constraints come from local sample points, either or both could be irrational. Discussion of the feature's usage can be found following Algorithm 31.

Figure 6-1 shows an example of a call to PartialCylindricalAlgebraicDecompose in Maple with passing of lifting constraints (deduced from an existentially quantified conjunction by SuggestCADOptions — Chapter 6). Only CylindricalAlgebraicDecompose (full CAD) and PartialCylindricalAlgebraicDecompose (CAD by QE with no VTS assistance) support passing of lifting constraints via the keyword option `'LiftingConstraints'`.

3.7 Equational Constraints in CAD

An optimisation for projection first suggested for the McCallum projection operator [48] allows usage of a restricted projection operator, meaning that fewer projection polynomials are produced in the bases for each variable, still allowing sufficient deduction of a quantifier free answer for QE but cutting down on the work for the step to which the majority of CAD's complexity can be attributed. The concept owes to the idea that for a formula of the form $\bigwedge(f = 0, \dots)$, examination of positive or negative sign of f is meaningless, and so sign invariance of f is only required in the sections of f.

Definition 44 (Equational Constraint). *An* equational constraint *(EC) of a Tarski formula Φ is a polynomial f from an equation $f = 0$ logically implied by Φ, explicitly or otherwise, for example $f = 0 \vee g = 0$ is equivalent to $fg = 0$, hence fg is an implicit equational constraint of $f = 0 \vee g = 0$. In a formula $\bigwedge(f_1 = 0, \dots, f_k = 0, \dots)$, $k > 1$, only one of the f_i, $1 \leq i \leq k$ may be designated as the equational constraint of the formula (at any one projection level).*

Some typical examples of implicit/explicit equational constraints are below, originally from [28] and typically used to describe a sample of implicit vs. explicit cases for ECs.

1. The formula $f = 0 \wedge g > 0$ has an explicit EC, $f = 0$.
2. The formula $f = 0 \vee g = 0$ has no explicit EC, but the equation $fg = 0$ is an implicit EC.

3. The formula $f^2 + g^2 \leq 0$ has no explicit EC, but it has at least two implicit ECs that are immediate: $f = 0$ and $g = 0$.

4. The formula $f = 0 \vee f^2 + g^2 \leq 0$ logically implies $f = 0$, and the equation is an atom of the formula which makes it an explicit EC according to the definition. However, since this deduction is semantic rather than syntactic, it is more like an implicit EC rather than an explicit EC.

`QuantifierElimination` would be able to deduce ECs in cases 1 and 2 only. In particular, Algorithm 21 can recursively collect an implicit equational constraint from a disjunction, but the other implicit examples are beyond the scope of the algorithm.

McCallum's first work on ECs [48] justified usage of a restricted operator for the first projection step, i.e. projection with respect to x_n. However [49] justified:

- usage of *multiple equational constraints*, justifying usage of a restricted operator for the first and last projection phases x_n and x_1, and semi-restricted operators otherwise, as opposed to a single equational constraint that would allow restricted projection only in x_n where appropriate, and

- *propagation of equational constraints*, giving further scope for selection of equational constraints amongst those that can now be identified as a result of the resultant rule (3.3) between equational constraints in previous variable(s).

Importantly, we may only select at most one equational constraint to "restrict" a projection operator with per level, i.e. iteration of projection. This is despite the potential existence of several of non trivial degree in that variable. Further, a selected equational constraint per level will henceforth be referred to as a "pivot", terminology again owing to McCallum and discussed further later. Naturally, propagation of equational constraints requires usage of multiple equational constraints to be relevant. Usage of multiple equational constraints means that equational constraints directly owing to the input formula in $x_2, \ldots, x_{n-1}$ could be selected for use in a (semi-)restricted projection operation, while propagation means that in addition those polynomials owing to the resultant rule (3.3) could be chosen as pivot in their stead.

For the first projection, we project from the polynomials appearing directly from the decomposition of the formula Φ. Bar equational constraints, we ignore boolean structure (at least in terms of $\wedge, \vee$) and require a flat representation of the polynomials, such as sets. Hence we require a recursive method to collect the polynomials from the relations of Φ. Because of equational constraints, we should pay attention to the polynomials of equations at the top level of Φ if Φ is a conjunction, but further because of the suggestion of implicit equational constraints arising from disjunctions of equations just below the top level conjunction as in Definition 44, this implies a more bespoke recursive collection of polynomials into a set of inequalities A and equations E, which is given as Algorithm 21.

The formulation of Algorithm 21 means that the deduced equational constraints are $f_1, \ldots, f_k$, $k \geq 0$ such that the input formula for CAD $Q_{n-m+1}x_{n-m+1} \ldots Q_n x_n$

Algorithm 21 Collection of all polynomials in a Tarski formula into a set of polynomials associated with inequalities, and a set of ECs

Input: Φ, a prenex unquantified real Tarski formula
Output: A, a set of polynomials associated to inequalities from Φ, and E a set of equational constraints from Φ

1: **procedure** GETPOLYSETS(Φ)
2: **if** $\Phi = \mathit{true}$ or $\Phi = \mathit{false}$ **then**
3: **return** $\emptyset$, $\emptyset$
4: **elseif** $\Phi = (f = 0)$ **then**
5: **return** $\emptyset$, $\{f\}$
6: **elseif** $\Phi = (f \,\rho\, 0)$ **then** ▷ $\rho \in \{<, \leq, \neq\}$
7: **return** $\{f\}$, $\emptyset$
8: **elseif** $\Phi = \bigvee(\Psi_1, \ldots, \Psi_k)$, $k > 1$ **then**
9: $(\ A,\ E,\ p\ ) \leftarrow \emptyset, \emptyset, 1$
10: **for** i **to** k **do**
11: **if** $\Psi_i = (f = 0)$ **then**
12: **if** $\deg(f) > 0$ **then**
13: $p \leftarrow pf$ ▷ Attempt to gather the implicit EC within a disjunction — item (2)
14: **end if**
15: **elseif** $\Psi_i = (f \,\rho\, 0)$ **then**
16: $A \leftarrow A \cup \{f\}$
17: **else**
18: $(\ A_{\text{out}},\ E_{\text{out}}\ ) \leftarrow$ getPolySets(Ψ_i)
19: $A \leftarrow A \cup A_{\text{out}}$
20: $A \leftarrow A \cup E_{\text{out}}$ ▷ Ψ_i must be a conjunction, and can't bring ECs up from a conjunction to a disjunction
21: **end if**
22: **end for**
23: **if** $|E| = 0$ **then**
24: **if** $p \neq 1$ **then**
25: **if** $|A| = 0$ **then**
26: **return** $\emptyset$, $E \cup \{p\}$
27: **else**
28: **return** $A \cup \{p\}$, E
29: **end if**
30: **else**
31: **return** A, E
32: **end if**
33: **elseif** $p = 1$ **then**
34: **return** $A \cup E$, $\emptyset$
35: **else**
36: **return** A, $\{pf \mid f \in E\}$
37: **end if**

Algorithm 21 Collection of all polynomials into sets of polynomials from inequalities and equational constraints, Part 2

```
38:     else                                          ▷ Φ = ⋀(Ψ_1, ..., Ψ_k), k > 1
39:         ( A, E ) ← ∅, ∅
40:         for i to k do
41:             if Ψ_i = (f = 0) then
42:                 E ← E ∪ {f}                       ▷ An explicit EC — item (1)
43:             elseif Ψ_i = (f ρ 0) then
44:                 A ← A ∪ {f}
45:             else
46:                 ( A_out, E_out ) ← getPolySets( Ψ_i )
47:                 A ← A ∪ A_out
48:                 E ← E ∪ E_out
49:             end if
50:         end for
51:         return A, E
52:     end if
53: end procedure
```

$\Phi(x_1, \ldots, x_n)$ is equivalent to

$$Q_{n-m+1}x_{n-m+1} \ldots Q_n x_n \bigwedge_{i=1}^{k} f_i(x_1, \ldots, x_n) = 0 \wedge \Phi_I(x_1, \ldots, x_n) \tag{3.2}$$

where m could here be 0 if passing an unquantified formula for production of a full CAD, and some of the f_i may have arisen from disjunctions directly below the top level conjunction, and so in essence such an $f_i = f_{i_1}(x_1, \ldots, x_n) = 0 \vee \ldots \vee f_{i_d}(x_1, \ldots, x_n) = 0$, $i_d > 1$ due to item (2) in examples of equational constraints at the beginning of this section. Again, other implicit equational constraints are out of the scope of what Algorithm 21 is able to deduce. If Φ_I is a conjunction, it should be free of equations amongst the top level operands such that $f_1, \ldots, f_k$ is the maximal such set of equational constraints (Algorithm 21 will never omit equational constraints from the top level of a conjunction).

Line 13 of Algorithm 21 suggests that we multiply any polynomials from equations when processing a disjunction because of the relevant "implicit equational constraint" condition (item 2 from the beginning of this Section 3.7). One may worry about the fact that we may have to attempt to factor these later in order to form square-free bases before projection, assuming they undergo no further processing. In terms of Maple, one may multiply these polynomials *without expansion*, and as such a call to factorisation *will* retain the factors. There is however, an interesting question of what happens if we are to preprocess these equational constraints with respect to a Gröbner basis beforehand (Section 3.7.3) — will we lose *all* information about the factorisation if we make such a basis?

Open Problem 45. *For a set of polynomials $f_1, \ldots, f_k$ for each of which we know of a partial factorisation $f_{i_1}, \ldots, f_{i_j}$, $i_j \geq 1$, $1 \leq i \leq k$, can the information about the partial factorisations be used at all by a Gröbner basis on $f_{i_1}, \ldots, f_{i_j}$? Is it dependent on monomial ordering?*

Definition 46 (Restricted Projection). *The restricted projection of a set of polynomials A with respect to a set of factors of one equational constraint E (a pivot set), for P the McCallum projection operator (P_M from Section 3.3) is defined by McCallum in [48] as:*

$$P_E(A) = P(E) \cup \{\text{res}_x(f, g) \mid f \in A,\, g \in E\}.$$

The semi-restricted projection of a a set of polynomials A with respect to a pivot set E is defined by McCallum in [48] as:

$$P_E^*(A) = P_E(A) \cup \{\text{discrim}_x(f) \mid f \in E\}$$

[57, Definition 11] formalises the restricted projection operator using the Lazard projection i.e. $P = P_L$, and in principle the semi-restricted projection operator is adapted to use P_L similarly.

`QuantifierElimination` implements the Lazard projection, and hence restricted projection via equational constraints with the Lazard projection is what is relevant further. Much of this is based on or concurrent with the research of [57, 56, 55]. Usage of equational constraints in CAD in `QuantifierElimination` is controlled by the keyword option `'UseEquations'`. Allowable values for this option are `'none'`, `'single'`, and `'multiple'`. Propagation of equational constraints in CAD in `QuantifierElimination` is controlled by the keyword option `'PropagateECs'`, which takes a *true* or *false* value. These options appear as arguments to essentially all top level routines using CAD at all in `QuantifierElimination`, including QuantifierEliminate, PartialCylindricalAlgebraicDecompose, CylindricalAlgebraicDecompose InsertFormula, etc. As a keyword option, all top level functions have to oblige a default value, which can be set to `'single'` for the release of `QuantifierElimination` to reflect the fact that only usage of a single equational constraint in x_n for Lazard CAD is guaranteed "complete" (see Section 3.7.2).

To explain propagation of equational constraints, via McCallum [49], we find the so-called "resultant rule":

$$f = 0 \wedge g = 0 \Rightarrow \text{res}_x(f, g) = 0 \tag{3.3}$$

and as such the resultant of two polynomials owing to equational constraints is an equational constraint for the next level. This is where we gain an important distinction — while the pivot set that we perform a restricted projection with respect to is a subset of all equational constraints at that level, we should track the remaining equational constraints that were *not* used as pivots, such that we can propagate this resultant rule and attempt use (semi-)restricted projection operators at successive

levels. Whenever we run out of potential equational constraints at a certain level, we stop usage of restricted projection operators entirely, and revert to standard projection.

Relevance of the resultant rule means the set $\{\text{res}_x(f,g) \mid f \in A,\, g \in E\}$ in the union to receive $P_E(A)$ of Definition 46 is only meaningful when examining any one projection set in isolation, and really we need to track up to three sets of polynomials at any one level — the (basis) set of pivots, C_i, $1 \leq i \leq n-1$, a basis of other equational constraints not used as pivots, B_{E_i}, $1 \leq i \leq n-1$, and a basis of all other polynomials or "inequalities", B_{A_i}, $1 \leq i \leq n$. In other words, programmatically, that set really needs to be calculated as the union

$$\{\text{res}_x(f,g) \mid f \in B_{E_i},\, g \in C_i\} \cup \{\text{res}_x(f,g) \mid f \in B_{A_i},\, g \in C_i\},$$

where the former operand is a set of potential equational constraints at level $i-1$ if propagation of ECs & hence the resultant rule is being used. This is because equational constraints not chosen as pivots are treated as non ECs, except we need to be cognizant they are still equational constraints for usage of the resultant rule. Algorithms 5 and 9 reflect this identification. This gives impetus to the characterisation of a `projection` object as discussed in Section 3.3, with visualisations of the relevant projection steps on these bases in Figures 3-1, 3-3 and 3-5.

3.7.1 Pivot Selection Strategy

Algorithm 22 represents the implemented strategy in `QuantifierElimination` for selection of a pivot at any one level. We must provide motivation as to why so much attention is given to the content of whatever polynomial is chosen as pivot, to the extent it is a return value of Algorithm 22. Let $c_1^{d_1} \cdots c_\zeta^{d_\zeta} \cdot p_{d_{\zeta+1}}^{d_{\zeta+1}} \cdots p_\xi^{d_\xi}$ be the factored representation of the equational constraint polynomial chosen as pivot, as it is formatted on line 5, such that

$$\bigvee_{k=1}^{\zeta} c_k^{d_k} = 0 \vee \bigvee_{i=\zeta+1}^{\xi} p_i^{d_i} = 0 \equiv \bigvee_{k=1}^{\zeta} c_k = 0 \vee \bigvee_{i=\zeta+1}^{\xi} p_i = 0$$

noting the multiplicities $d_1, \ldots, d_\xi$ of all the factors are meaningless for these equations. Now, let f be any other equational constraint polynomial from E (that will find itself in factored form in E_F as a result of Algorithm 22. Then, in restricted projection with respect to the Lazard operator P_L, as part of the set of resultants of polynomials from the pivot set with all other polynomials in x_j (including those from E_F), we have

$$\text{res}_{x_j}\left(\prod_{i=\zeta+1}^{\xi} p_i, f\right) = 0 \vee \bigvee_{k=1}^{\zeta} c_k = 0,$$

or equivalently via properties of resultants

$$\prod_{i=\zeta+1}^{\xi} \text{res}_{x_j}(p_i, f) = 0 \vee \bigvee_{k=1}^{\zeta} c_k = 0,$$

Algorithm 22 Selection of a pivot set from a set of equational constraints

Input: E, a set of polynomials that are equational constraints at some level in projection, and x_j, the variable to project with respect to next. E is assumed to contain at least one polynomial of non trivial degree in x_j.

Output: E_P, the output pivot set to use, E_F the set of equational constraints not chosen as pivot in factored form, E_C, all equational constraints of degree 0 in x_j, and $\mathrm{c_{out}}$, the content of the chosen pivot (degree 0 in x_j) in factored form

1: **procedure** CHOOSEPIVOTSET(E, x_j)
2: ($\mathrm{d_{best}}$, $\mathrm{n_{best}}$, $\mathrm{d^c_{best}}$, $\mathrm{n^c_{best}}$) $\leftarrow \infty, \infty, \infty, \infty$
3: **for** p **in** E **do**
4: **if** $\deg_{x_j}(p) > 0$ **then**
5: Fully factor p as $c_1^{d_1} \cdots c_\zeta^{d_\zeta} \cdot p_{\zeta+1}^{d_{\zeta+1}} \cdots p_\xi^{d_\xi}$ where $\deg_{x_j}(c_i) = 0$, $1 \leq i \leq \zeta$, $\deg_{x_j}(p_i) > 0$, $\zeta + 1 \leq i \leq \xi$, $d_i > 0, 1 \leq i \leq \xi$
6: $E_F \leftarrow E_F \cup \{p_{\zeta+1} \cdots p_\xi\}$ ▷ Add factors of p of non trivial degree in x_j to E_F in factored form
7: **if** p is the best equational constraint polynomial observed from E in terms of the tiebreakers 1 through 4 discussed in this subsection, in light of the values of $\mathrm{d_{best}}$, $\mathrm{n_{best}}$, $\mathrm{d^c_{best}}$, $\mathrm{n^c_{best}}$ to compare **then**
8: Update $\mathrm{d_{best}}$ as the maximum degree of any factor of p in x_j
9: Update $\mathrm{n_{best}}$ as the number of factors of p of non trivial degree in x_j
10: Update $\mathrm{d^c_{best}}$ as the maximum degree of any factor of content of p in x_j
11: Update $\mathrm{n^c_{best}}$ as the number of factors of p of degree 0 in x_j
12: $\mathrm{c_{out}} \leftarrow c_1 \cdots c_\zeta$ ▷ The content of p in x_j, in factored form
13: $E_P \leftarrow \{p_{\zeta+1}, \ldots, p_\xi\}$ ▷ Best potential pivot set — factors of p of non trivial degree in x_j
14: **end if**
15: **else**
16: $E_C \leftarrow E_C \cup \{p\}$ ▷ With no factorisation
17: **end if**
18: **end for**
19: $E_F \leftarrow E_F \setminus \{\prod E_P\}$
20: **return** E_P, E_F, E_C, $\mathrm{c_{out}}$
21: **end procedure**

which via the logic of equivalence of a disjunction of equations to one equation (such as (2) in examples of ECs)

$$\prod_{k=1}^{\zeta} c_k \prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f) = 0$$

and so we return $\prod_{k=1}^{\zeta} c_k$ from Algorithm 22 such that it can be passed to restricted Lazard projection, instead of multiplying every $f \in E_F$ through by the content of the pivot, which will be explained to be undesirable, albeit feasible.

If we are using propagation of equational constraints, this applies for every $f \in E_F$, such that in restricted projection with respect to the Lazard operator our next set of candidate equational constraints with respect to x_{j-1} (if $j > 2$) is

$$\bigcup_{f \in E_F} \left(\prod_{k=1}^{\zeta} c_k \prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f) \right) \cup E_C. \tag{3.4}$$

Note that via properties of resultants,

$$\prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f \prod_{k=1}^{\zeta} c_k) = 0 \Rightarrow \prod_{k=1}^{\zeta} c_k \prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f) = 0,$$

but actually

$$\prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f \prod_{k=1}^{\zeta} c_k) =$$

$$\prod_{i=\zeta+1}^{\xi} \left(\prod_{k=1}^{\zeta} c_k \right)^{\max(\deg_{x_j}(f), \deg_{x_j}(p_i))} \operatorname{res}_{x_j}(p_i, f) =$$

$$\left(\prod_{k=1}^{\zeta} c_k \right)^{\sum_{i=\zeta+1}^{\xi} \max(\deg_{x_j}(f), \deg_{x_j}(p_i))} \prod_{i=\zeta+1}^{\xi} \operatorname{res}_{x_j}(p_i, f)$$

where we know that $\max(\deg_{x_j}(f), \deg_{x_j}(p_i)) > 0 \ \forall i \in \{\zeta + 1, \ldots, \xi\}$ via the construction of the pivot set and E_F (so we would not lose the contents), but introducing some undesirable degree via the exponent (even if we receive something still in factored form), so we do not brazenly multiply every element of E_F in Algorithm 22 through by $\mathrm{c_{out}}$ in order to receive the new set of equational constraints later, even if it would be convenient.

Having investigated what will be done with the content of the chosen pivot upon its selection, we can begin to explain the tiebreaker conditions within Algorithm 22. Written in plain English, they aim to select an equational constraint of non trivial degree in x_j that has:

1. The lowest maximum degree factor amongst those of non zero degree in x_j (the $p_{\zeta+1}, \ldots, p_\xi$),
2. The least numerous factors amongst those of non zero degree in x_j (again, the $p_{\zeta+1}, \ldots, p_\xi$),
3. The lowest maximum degree factor amongst those of degree zero in x_j (the $c_1, \ldots, c_\zeta$),
4. The least numerous factors amongst those of degree zero in x_j (again, the $c_1, \ldots, c_\zeta$).

Lemma 47. *Usage of Algorithm 22 achieves the goal of being a "greedy" pivot selection strategy, with respect to the polynomial operations to occur in restricted Lazard projection with respect to x_j after its selection.*

Proof. (3.4) represents the set of potential equational constraints in x_{j-1}, assuming $j > 2$. The union across E_F corresponds precisely to the contribution from propagation of equational constraints, while E_C is the set of any equational constraints which were immediately of degree 0 in x_j. It is henceforth operations owing to this set, either directly or indirectly, that we attempt to minimise.
Putting tiebreaker 1 before 2 addresses the notion that the total complexity of the ensuing resultants in P_L are more dependent on input degree than how numerous they are.
Tiebreakers 3 and 4 are technically irrelevant to the situation in x_j, but attempt to do some work to minimise the degree of potential equational constraints in any further restricted projection steps, again assuming propagation of equational constraints. If we are not propagating equational constraints, or a pivot from E_C is to be selected instead, the penalty of examination of these criteria is minimal and tiebreakers 1 and 2 were still superior. Again referring to (3.4), we attempt to minimise $\max_{k=1}^{\zeta} \deg(c_k)$, or in failing to distinguish between those maxima, ζ. This is pertinent as $\prod_{k=1}^{\zeta} c_k$ multiplies every polynomial in that union, potentially making the certain resultants to occur as a result of selection of any of those polynomials as pivot more costly.

[5] discusses selection of a pivot (designation of an equational constraint) in usage of a single EC with the McCallum projection using the sotd or ndrr metrics on generated projection bases, which is tied to variable strategy (Section 3.8). Variable strategy and pivot selection largely interact differently in `QuantifierElimination`, and in particular those metrics are not used to select pivots, with Algorithm 22 always applied instead.

3.7.2 Curtains in a Lazard projection CAD

[57] introduces the concept of "curtains". These mathematical obstacles may occur when using equational constraints with the Lazard projection in CAD, and may impede our ability to construct (enough of) a Lazard invariant, and hence sign invariant CAD, for whatever purpose the CAD is to perform.

Definition 48 (Variety). *[21, Definition 50] The set of solutions over K (a field) of an ideal I is called the variety of I.*

The context for I further is that formed by (at most all) equational constraints from input for some CAD for Lazard projection (with restricted projection operations arising from said equational constraints).

Definition 49. *[57, Definition 8] A variety $C \subseteq \mathbb{R}^n$ is called a curtain if, whenever $(\underline{x}, x_n) \in C$, then $(\underline{x}, y) \in C$ for all $y \in \mathbb{R}$.*

Definition 50. *[57, Definition 9] Suppose $f \in \mathbb{R}[x_1, \ldots, x_n]$ and $W \subseteq \mathbb{R}^{n-1}$. We say that f has a curtain at W if for all $\underline{x} \in W$ and $y \in \mathbb{R}$ we have $f(\underline{x}, y) = 0$.*

Definition 51 (Well Oriented). *[50, Remark 2, Algorithm CADR3] A set A of r-variate polynomials over $\mathbb{R}$ is said to be* well-oriented *if no element of* $\text{prim}(A)$ *vanishes identically on any submanifold of $\mathbb{R}^{r-1}$ of positive dimension and, moreover, this property holds recursively for the set* $\text{cont}(A) \cup P(B)$*, where B is the finest square-free basis for* $\text{prim}(A)$*.*

Remark 52. *In Definition 51 above, P is assumed to be the McCallum projection operator P_M, and r is what would normally contextually be $n \in \mathbb{N}$ in this work.* $\text{prim}(A)$ *is the set of primitive parts of all polynomials in A, and* $\text{cont}(A)$ *the set of contents of all polynomials in A, both with respect to one variable under a fixed ordering. A is most relevant when taken to be the flat top level set of all polynomials from input.*

Proposition 53. *[57, Proposition 4] Let $f \in \mathbb{R}[x_1, \ldots, x_n]$ be of positive degree in x_n and let S be a connected subset of $\mathbb{R}^{n-1}$. Suppose that f is Lazard delineable on S. Then f is Lazard (i.e. lex-least) invariant in each Lazard section and sector of f over S.*

Lack of well orientedness (Definition 51) may lead to nullification occurrences in use of the McCallum projection operator P_M. In other words, the input polynomials can retrospectively be seen to not be "well-oriented" when some projection polynomial is nullified after evaluation at the sample point of some cell of level less than n in stack construction. This evaluation is the equivalent of Lazard evaluation (Algorithm 11), except usage of Lazard evaluation in conjunction with the Lazard projection avoids this issue on non pivot polynomials by removing nullifying factors. With standard evaluation via sample points, nullification obfuscates the potentially changing sign(s) of polynomial(s) on that cell, as the evaluation is 0 instead of any non trivial polynomial potentially donating roots to build around. However, Lazard evaluation rids us of this issue by dividing through by nullifying factors. Hence `QuantifierElimination` does not suffer from nullification, but suffers from a related issue due to equational constraints.

Definition 54 (Lazard Curtain). *A* Lazard curtain *is a subset of the space formed by a CAD cell c of level $0 < k \leq n-1$ such that there exists an f in the pivot set of level $k+1$, C_k where the full sample point $\alpha \in \mathbb{R}^k$ of c nullifies f, where the set C_k exists. If C_k doesn't exist, i.e. there were no equational constraints used at level k, then there are certainly no CAD cells with curtains of level k.*

In terms of Definition 50, f is the multiplication of all polynomials in that pivot set (or implicitly any of them due to multiplication of the factors), and $W \subseteq c$. α nullifying

some element f of the pivot set C_k is equivalent to f having a non zero Lazard valuation on α (and hence implicitly c) (Algorithm 23).

In `QuantifierElimination`, nullification occurrences are only relevant on polynomials from pivot sets, and are characterised as "Lazard curtains" — the further interest of this section. Where a cell c has a curtain on a pivot polynomial at level $2 \leq i \leq n$, Lazard evaluation is insufficient because the lack of cross resultants between polynomials in the relevant sets of polynomials treated as those from inequalities B_{A_i} and $B_{E_{i-1}}$ due to usage of a (semi-)restricted projection operator may mean we will fail to identify all the geometry in subsequent lifting stages that allow us to decompose c to sign invariance over that pivot polynomial. More precisely, if $f, g \in B_{A_i} \cup B_{E_{i-1}}$, and $h \in C_{i-1}$, then on a curtain, $r_1 \coloneqq \text{res}_{x_i}(f, h) = 0$, and $r_2 \coloneqq \text{res}_{x_i}(g, h) = 0$, hence $\text{res}_{x_{i-1}}(r_1, r_2) = 0$[2], and hence the polynomial intended to deduce the finer geometry to achieve sign invariance cannot do so. In contrast to usage of the McCallum projection, we must pay attention to whether polynomials from pivot sets nullify, as opposed to all projection polynomials.

These Lazard curtains impede our ability to construct a Lazard & hence sign invariant CAD, and so (at the least) identifying, and otherwise avoiding or dealing with them is of interest, e.g. because sign invariance implies truth invariance on the top level formula for QE by Partial CAD. Nullification occurrences in the context of the McCallum projection in situations lacking well-orientedness are also curtains, and in retrospect could be called "McCallum curtains" in the context of the nomenclature introduced here. Similarly, those curtains impede the ability to construct a sign invariant CAD, and so `QEPCAD B` or `ProjectionCAD` identify nullification occurrences in various ways which will be discussed later in comparison with what is implemented and available in `QuantifierElimination` for handling of curtains with the Lazard projection. Again, McCallum nullification occurrences are not specific to usage of equational constraints, in contrast to Lazard curtains.

CAD in `QuantifierElimination` will check for a curtain on any cell c that undergoes stack construction, that is via CCHILD. At this point, we check for a non zero Lazard valuation of any f in the pivot set commensurate with the level of c, if such a pivot set exists. Hence this depends on the extent to which equational constraints were used, i.e. single or multiple equational constraints (controlled by the CAD keyword option `'UseEquations'`). For a cell c of level $0 < k \leq n-1$, we examine the pivot set C_k, which is actually the set of factors of an equational constraint for x_{k+1}. If this pivot set is not a singleton, this means whatever equational constraint was chosen as a pivot factored, and so nullification of any of its factors nullifies the original polynomial chosen. Due to the likelihood that usage of at least a single equational constraint is enabled, C_{n-1} is likely to exist if any equational constraints were identified in x_n. Checking the valuations of the pivot set on c before continuing with the rest of stack construction via CCHILD is important, because the function univariateBasisAtLazard used in

[2]If $f, g \in B_{E_{i-1}}$, then at most one of r_1 and r_2 could be identified as a pivot at level $i-1$, so $\text{res}_{x_{i-1}}(r_1, r_2)$ exists via the ensuing cross resultants in (semi-)restricted projection.

CCHILD merely iterates over all polynomials at a canonical projection level performing Lazard evaluations on each, making no distinction as to a projection polynomial's origin. Lazard valuations (not to be confused with *evaluation*) are not usually purposefully identified during Lazard evaluation (Algorithm 11) in `QuantifierElimination`, although in practice they are a corollary of computation of evaluation. Here, even the actual values in requested valuations are meaningless beyond "non zero", so the algorithm for detection of a non zero valuation and hence curtain on one cell is bespoke, to avoid the verbosity of calling Algorithm 11 (or a modification thereof).

Algorithm 23 Check for a non zero Lazard valuation on a `CADCell`

Input: c a `CADCell` of level $0 < k < n$, $f \in \mathbb{R}[x_1, \ldots, x_{k+1}]$ an element of some pivot set C_k used in projection — hence of non trivial degree in x_{k+1}
Output: *true*, if the Lazard valuation of f on α, the sample point of c, is non zero, else *false*

1: **procedure** checkLazardValuationNonZero(c, f)
2: $\alpha \leftarrow$ GetSamplePoints(c) ▷ Full sample point of c
3: $f' \leftarrow f$
4: **for** i **to** k **do**
5: $s \leftarrow \text{lhs}(\alpha_i) - \text{rhs}(\alpha_i)$ ▷ s becomes the ith element of the full sample point as a linear polynomial
6: **if** $s \mid f'$ **then**
7: **return** *true*
8: **else**
9: $f' \leftarrow f'|_{\alpha_i}$ ▷ still Lazard valuation, but no division beforehand as if there were any we've already returned *true*
10: **end if**
11: **end for**
12: **return** *false*
13: **end procedure**

Note that α_i is the ith element of the full sample point of c, i.e. an equation defined by a real linear polynomial in x_i. $\text{lhs}(\alpha_i) - \text{rhs}(\alpha_i)$ is that linear polynomial. Usage of Algorithm 23 is evidently less verbose than doing the full valuation and checking that it is non zero, i.e. along the lines of Algorithm 11.

The work of [56] goes further as to prove that "point curtains" are of no cause for concern.

Definition 55 (Point Curtain). *[56, Definition 13] We say that $f \in \mathbb{R}[x_1, \ldots, x_n]$ has a point curtain at $\alpha \in \mathbb{R}^{n-1}$ if the Lazard valuation (Definition 36) of f on α $\nu_\alpha(f) \neq (0, \ldots, 0)$ and there exists a neighbourhood U of α such that $\nu_{\alpha'}(f) = (0, \ldots, 0)$ for all $\alpha' \in U \setminus \{\alpha\}$*

Theorem 56. *[56, Theorem 3] Let $f \in \mathbb{R}[x_1, \ldots, x_n]$ and let $\alpha \in \mathbb{R}^{n-1}$. If f is an equational constraint and has a point curtain at α, then PL_E is sufficient to obtain a*

sign-invariant CAD.

Remark 57. *Theorem 56 means that if we construct a Lazard CAD such that the only level $n-1$ curtains are point curtains, then we can actually achieve sign invariance. This does not imply Lazard invariance, but in general Lazard invariance is merely to imply sign invariance, e.g. giving truth invariance to achieve QE.*

Definition 58 (1-Cell Neighbours). *If c is a level $k > 0$ cell with full index $[i_1, \ldots, i_k]$, its 1-cell neighbours are the cells with full index $[i_1, \ldots, i_j - 1, \ldots, i_k]$ and $[i_1, \ldots, i_j + 1, \ldots, i_k]$, for each $1 \leq j \leq k$, where those cells exist.*

In the context of that work, a CAD cell and its (full) sample point α are used interchangeably, which is entirely canonical considering the bijection between full sample points and cells. [56] refers to "1-cell neighbours" (Definition 58) to mean the neighbourhood U from Definition 55. Algorithm 24 can be used to get an `Array` of all the 1-cell neighbours found for a cell, such that one can begin to deduce if a certain cell is a point curtain or not. Algorithm 26 is a canonicalization that a non zero Lazard valuation of any factor of the top level pivot set implies a non zero valuation on the pivot (because the valuation of a product is the sum of the valuations of its factors), and provides an interpretation of this condition in terms of the `projection` object of `QuantifierElimination`.

Algorithm 24 Algorithm to get 1-cell neighbours of a `CADCell`

Input: c a level $k > 0$ `CADCell` to find 1-cell neighbours of
Output: N, a (possibly empty) `Array` of the 1-cell neighbours of c (Definition 58)

procedure GETONECELLNEIGHBOURS(c)
  $[i_1, \ldots, i_k] \leftarrow$ getFullIndex(c) ▷ Full cell index for c
  cell $\leftarrow c$
  $N \leftarrow$ an empty `Array`
  for j **from** k **down to** 1 **do**
    cell $\leftarrow$ cell $\mapsto$ parent
    $N \leftarrow N \,\|\, \text{findChildAt(}\ \text{cell}, [i_1, \ldots, i_j - 1, \ldots, i_k], j\ \text{)}$
    $N \leftarrow N \,\|\, \text{findChildAt(}\ \text{cell}, [i_1, \ldots, i_j + 1, \ldots, i_k], j\ \text{)}$
  end for
  return N
end procedure

Algorithm 24 and its helper Algorithm 25 attempt to create a container of all one cell neighbours of a `CADCell` by tree traversal.

1. This can be used to find neighbours of a cell of any level, even if the likely intention is that we return neighbours of level $n-1$ cells given the later discussion of this section enabled by Theorem 56.

2. Algorithm 24 traverses up the tree, while Algorithm 25 traverses down. We use the fact that the level $k > 0$ cell c and its neighbours with indices $[i_1, \ldots, i_j -$

Algorithm 25 Algorithm to find a `CADCell` of a specific index from a cell of a specific level

Input: c a `CADCell` of level $0 \leq d \leq n$, ind a list with $\leq n$ (likely $n-1$, or at least $\leq n-1$) elements representing the index of the cell in the subtree below c to find and ideally return, j a positive integer such that $1 \leq j \leq n$

Output: If a cell with full index ind could be found in the subtree below c, then that cell, else nothing

```
1:  procedure findChildAt( c, ind, j )
2:      i ← j
3:      temp ← c
4:      repeat
5:          if temp has children cells (temp ↦ children) then
6:              childs ← [e_1, ..., e_t], where each e_m is the local indices of the mth child
                    cell of temp, where there are t > 0 such child cells.
7:              k ← Search( childs, ind[i] )
8:              if k = 0 then
9:                  return                          ▷ Return e.g. NULL — no cell found
10:             else
11:                 temp ← c ↦ children[k]
12:                 i--
13:             end if
14:         else
15:             return
16:         end if
17:     until i = 0
18:     return temp
19: end procedure
```

$1, \ldots, i_k]$ and $[i_1, \ldots, i_j + 1, \ldots, i_k]$ are contained in the CAD subtree rooted at the cell $k-j+1$ levels up from c, hence we iteratively look upwards in the tree and search within the smallest necessary subtree with height commensurate with the element of the index that we are searching for, rather than needlessly searching beneath the root cell every time.

3. Line 7 is characterised as fairly generic — the intention is that Search has the semantics that it returns 0 if the second argument could not be found in the first argument (a list), else it returns the index where the value was found. `QuantifierElimination` requires that child cells in the children `Array` for a `CADCell` are stored in increasing order of local index, but it is not certain that $[e_1, \ldots, e_t] = [1, \ldots, t]$, if any cells are missing due to (possibly evolutionary) usage of OpenCAD. Hence one can (generally) only justify usage of $\mathcal{O}(\log t)$ binary search amongst that list to find the index, else usage of Search(...) on line 7 can merely be replaced by $k \leftarrow \text{ind}[i]$, i.e. when search is superfluous because the local indices form the continuous list $[1, \ldots, t]$. One notes that binary search is not the main constituent of the headline complexity of the gathering of the neighbour cells, let alone the parent routines.

Algorithm 26 Algorithm to detect a Lazard curtain on a `CADCell`

Input: c a `CADCell` of level $0 < k < n - 1$, P a `projection` object
Output: *true*, if the Lazard valuation of any element of the pivot set C_k of level $k + 1$ from P is non zero, else *false*

```
1: procedure DETECTLAZARDCURTAIN( c, P )
2:     if P ↦ pivotSet[k] is a set of polynomials, i.e. a pivot set exists at level k + 1
           then
3:         for f in P ↦ pivotSet[k] do
4:             if checkLazardValuationNonZero( c, f ) then
5:                 return true
6:             end if
7:         end for
8:     end if
9:     return false
10: end procedure
```

Altogether, the checking for curtains with preceding conditions in Algorithm 14 (CCHILD) is in Fragment 27.

The first conditions in the "If" ensure, in order, that checking for curtains is desirable by the boolean curtainCheck, any projection occurred at all via checking that $n > 1$, and c is not the root cell and hence has a sample point. Further, if c has a non zero Lazard valuation on the relevant pivot set, then we produce an error about a Lazard curtain with the relevant data about the cell and equational constraint. Production of an error from CCHILD can and will be caught, as will be delineated later as part of a greater scheme to ignore "problematic" cells (especially in the context of QE) via the standard CAD lifting loops used throughout the various CAD routines in

Fragment 27 Checking for Lazard curtains before stack construction

```
1: if curtainCheck and n > 1 and c ↦ level > 0 and detectLazardCurtain( c, P )
       then
2:     Produce an ERROR about a Lazard curtain on c via the nullified pivot
           polynomial from P, to be caught by the function calling stack
           construction (e.g. line 6 of Code Fragment 28)
3: end if
```

`QuantifierElimination`.

Theorem 56 says that if c, the cell to construct a stack over, is merely a point curtain, i.e. its neighbours all have zero Lazard valuations, then stack construction can continue. However, we may not have constructed the neighbours of c, and so are unable to be certain that we identify a point curtain during regulation lifting. This is especially true when cell selection strategy has us traversing the tree depth-wise in Partial CAD, but is true regardless considering any one cell of a particular level has to be parsed by CCHILD first. Therefore we produce the error and it is handled by the top level in a context specific to the purpose of the CAD construction, which is especially pertinent considering we later discuss a method of recovery for a subset of such curtains. We can of course provide an upper bound for how many neighbour cells a level $k \geq 0$ cell can have — $2k$ via the characterisation of neighbour cells in Definition 58. This bound can be made sharper for any one cell in terms of how many members of the cell's full index $[i_1, \ldots, i_k]$ are extremities (i.e. 1 or the maximum such i_j amongst child cells of the cell with full index $[i_1, \ldots, i_{j-1}]$). If there are $0 \leq t \leq k$ such extremities, then there are $k \leq 2k - t \leq 2k$ neighbour cells. The number of extremities t for a `CADCell` is realised by Algorithm 33. This bound can be used to clarify that the set of neighbour cells found by Algorithm 24 is complete to be able to use the criteria about point curtains characterised by one-cell neighbours with confidence. Point curtains can be lifted normally via Theorem 56, so the classification is worthwhile. However, this classification is still delayed until as late as possible, for at least the following reasons:

- In Partial CAD, we may be able to lift to meaningful cell(s) deducing a quantifier free equivalent without engaging with the unnecessary expense of using the point curtain criteria on cells eventually deduced as not meaningful for output.
- If we let regulation lifting for CAD in any context finish, then we can be far more confidence about meeting the implied bound above for the numbers of neighbour cells for cells with curtains, instead of pointlessly traversing the tree to generate all neighbour cells to attempt to meet the bound.
- In the context of QE, we can then classify and/or "fix" only those cells which are absolutely necessary for output.

Lemma 59 consolidates the discussion on bounds for number of neighbour cells above.

Lemma 59 (Bound to achieve confidence in point curtain criteria). *If we generate all neighbour cells for a level $0 < k < n$ `CADCell` c with a curtain (e.g. via*

Algorithm 24), and we meet the bound $2k-t$ *where* $0 \leq t \leq k$ *is the number of extremal values in the full index* $[i_1, \ldots, i_k]$ *for* c*, then we can use the criteria on neighbour cells to identify* c *as a point curtain with confidence. If* N *is the set of all neighbour cells found for* c*, then*

$$|N| = 2k - t \wedge \neg \left(\bigvee_{c_N \in N} \text{detectLazardCurtain}(c_N, P) \right)$$

is the condition allowing us to confidently classify c *as a point curtain, where* P *is the* `projection` *object associated with the CAD containing* c*.*

Definition 60 (Low Level Curtain). *A* low level curtain *is one on a cell of level* $0 < k < n-1$*. The level of a curtain is of course contextual for the value of* n *with respect to the relevant CAD.*

Working further through [56], we find methodology to directly deal with non point curtains of level $n-1$ via further decomposition. Algorithm 3 from [56] provides this decomposition. One notes that the presentation there is strictly to provide a CAD of the curtains, however here it is of interest to *modify* an existing CAD such that it is sufficiently sign invariant and, where appropriate, deduce the quantifier free equivalent of a QE problem. The decomposition algorithm requires, at least for reasons of efficiency, *all* curtain cells to be accessible in a container beforehand before processing. Hence we can finally arrive at Code Fragment 28 describing lifting in Partial CAD (for QE, PartialCylindricalAlgebraicDecompose).

Level $n-1$ non point curtains may not be all that remains in terms of curtains during lifting, due to non point curtains of level $1 \leq k < n-1$, which would require different methodology to deal with. The methodology provided here *may* be able to work around such curtains in QE via Partial CAD, but not via full or "stock" CAD, at which point errors must be produced to terminate production of the CAD. However, curtains of level $1 \leq k < n-1$ are only relevant when multiple equational constraints are used via passing of the keyword option `'UseEquations' = 'multiple'`, and so a full CAD can certainly be produced either under the same variable ordering by passing `'UseEquations' = 'single'` instead, and here the only possible pivot set is one for x_n, at which point curtain decomposition can guarantee successful termination.
Code Fragment 28 uses the following:

1. `CAD_EXCEPTION_STRINGS` should be a sequence of strings for errors to catch that can occur in the context of CCHILD including, but not limited to, Lazard curtains. Other examples of errors that we may wish to be caught include those that can occur in the context of stack construction or cell evaluation, such as root isolation for the former — e.g. being unable to isolate roots of a convoluted polynomial given reasonable resources (see below).

2. We catch errors about any Lazard curtain, including curtains of level $k < n-1$, and more generally any error where `QuantifierElimination` could not otherwise

Fragment 28 Partial CAD (for QE) lifting loop in the context of "lifting failures"

```
1: Initialise problemCells as an empty container
2: while ( i ← QECADStrategy( cad ) ) > 0 do
3:     cell ← pop( cad, i )
4:     try
5:         CCHILD( cell, cad, bases, vars, n, localopen, constraints, bounds,
               curtainCheck = true )
6:     catch CAD_EXCEPTION_STRINGS:
7:         Append the list [cell, E] to problemCells, where E is the exception caught
8:         next
9:     end try
10:    PRPTV( cell, quants, m, n, cad, leaves, problemCells )
11: end while
```

recover, because the intent is any such problematic cells may be removed from the CAD tree via propagation of truth values.

3. problemCells is a container (presumably a `QEContainer`) containing cells stored together with a corresponding lifting error. Via the above it contains at least, and possibly at most, cells with Lazard curtains. Hence we can deduce a container of all curtains to decompose via the technology from [56].

4. problemCells is passed as an argument to PRPTV (Algorithm 15). The spirit of Partial CAD, and in particular PRPTV is that one should only lift cells where necessary to deduce a quantifier free equivalent of a QE problem. Cells that are due to receive stack construction exist in cad. Cells with lifting errors exist in problemCells, but we know that we originally aimed to build a stack over them. If truth value propagation leads us to remove a CAD subtree that includes a cell in problemCells, then we should reflect its removal from the subtree by removal from problemCells. Later, at least when it comes to decomposition of cells with curtains, we only decompose curtain cells where we know it is meaningful to do so to achieve QE.

Avoidance of General Lifting Failures

Importantly, Code Fragment 28 is able to attempt to ignore *any general failures* to construct or evaluate the child cells of a cell entering CCHILD or PRPTV. PRPTV includes its own try/catch mechanisms to trap errors arising from item 5, extending the avoidance of errors under the umbrella of Fragment 28. The full list of such failures that can be caught within this framework is as follows:

1. Lazard curtains, as is the main purpose of this section,

2. Being unable to isolate the roots of a convoluted polynomial given reasonable resources, which may manifest as "being unable to compute a root bound" (Section 3.4.1),

3. Being unable to make disjoint two root descriptions from separate lifting polynomials due to requirement of usage of a precision exceeding a set threshold (Section 3.4.1),

4. Being unable to evaluate any lifting constraint with irrational coefficients at a root of a lifting polynomial also with irrational coefficients in order to deduce whether the isolating interval about this root description is discarded before stack construction (Section 3.6),

5. Failure to deduce the sign of a relation on a constant expression of real algebraic numbers (specifically interval indexed RootOfs) during evaluation of a `CADCell` by evaluateTFArrayAtSP (Section 3.4),

6. Production of reducible RootOfs, i.e. RootOfs that imply ambiguity in any algebraic expression featuring polynomials (such as in a formula to evaluate, or polynomial to isolate roots of), which can occur in the current methodology of evaluating truth of relations of nested real algebraic numbers in evaluateTFArrayAtSP.

Hence `CAD_EXCEPTION_STRINGS` could be a sequence of strings containing at most all the items above. As the support within various routines used by `QuantifierElimination`'s CAD became more extensive during the development of `QuantifierElimination`, errors of the nature of 2, 3, or 4 became much less relevant, or at least far more rare, but the rationale for a framework to attempt to avoid and recover from such errors remains at least due to Lazard curtains, which is a mathematical complication rather than one of software. In particular, the author is grateful for continued support by Maplesoft & collaborators to improve root isolation & refinement for polynomials over `QuantifierElimination`'s representation of real algebraic numbers to mitigate the classes of error regarding root isolation (2, 3, and 4). The intention of usage of this code fragment is that we may be able to avoid lifting errors where propagation of truth values from other meaningful leaf cells deduces the nonnecessity of cells with lifting errors for output, hence their (implicit) removal from the CAD tree. The capture of general errors is hence only relevant in Partial CAD. Their storage is such that we have a mechanism for identifying their removal from the CAD tree, possibly reraising an exception if we cannot deduce QE, and recovery from a subset of curtain occurrences (explained shortly in this section).

As an example of lifting failure avoidance, we can discuss an example where propagation of a truth value allows us to ignore a Lazard curtain. Consider the fully existentially quantified example

$$\exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \exists x_7 \exists x_8 \exists x_9 \exists x_{10} \exists x_{11} \exists x_{12}$$
$$v_2 = 0 \wedge v_4 = v_3 \wedge v_9 = 1 \wedge v_{10} = 1 \wedge v_1\, v_9 + v_{11}\, v_5 = v_7 \wedge v_{11}\, v_6 + v_2\, v_9 = v_8 \wedge$$
$$v_{10}\, v_3 + v_{12}\, v_5 = v_7 \wedge v_{10}\, v_4 + v_{12}\, v_6 = v_8 \wedge 0 < v_{12} \wedge v_5 < 0 \wedge v_7 < 0 \wedge v_{11} < 0 \wedge$$
$$v_7 < 2\, v_8 \wedge v_8 < 0$$

which is the existentially quantified example associated to "Supply-Demand: Krugman scenario error 0013" from the Economics QE database (Section 7.1). In traversal of QE purely by Partial CAD with default options for PartialCylindricalAlgebraicDecompose and multiple ECs via `'UseEquations' = 'multiple'`, we find a Lazard curtain at a level 8 cell with full sample point

$$[v_9 = 1, v_{10} = 1, v_6 = 0, v_4 = 0, v_3 = 0, v_2 = 0, v_5 = -1, v_{12} = 1]$$

on the EC $v_{11}v_6 - v_{12}v_6 - v_3$. While the problem is essentially linear in each variable, usage of restricted projection (in various ECs with multiple equational constraints) allows us to omit various coefficients from polynomials owing to inequalities in projection (in the associated benchmarks, one notes the example only completes before timeout for QE with multiple ECs enabled). One notes that the curtain is at level $8 < 11 = 12 - 1 = n - 1$, which is important considering the attention paid to level $n - 1$ curtains soon in this section, because these are the only curtains that can be recovered from, as opposed to merely ignored/avoided. Because the problem is completely existentially quantified, we need only find a meaningful truth value at any cell to terminate QE. It happens that we find one from the cell with full sample point

$$[v_9 = 1, v_{10} = 1, v_6 = 1, v_4 = -1, v_3 = -1, v_2 = 0,$$
$$v_5 = -1, v_{12} = \frac{1}{2}, v_{11} = -\frac{1}{2}, v_8 = -\frac{1}{2}, v_7 = -\frac{3}{2}, v_1 = -2],$$

which of course propagates the truth value *true* all the way to the root, and the curtain found previously is superfluous.

Meanwhile, in the context of regular full CAD construction (CylindricalAlgebraicDecompose) without evaluation of truth values, we can obtain the similar Code Fragment 29. In full CAD, we need to construct every single level n cell. Lifting failures other than level $n - 1$ curtains cannot be recovered from due to lack of propagation of truth values, so the associated exceptions shouldn't be caught, unlike the case for Code Fragment 28. A subset of curtains can be recovered from (level $n - 1$ non point curtains), so we catch "Lazard curtain" errors, and deduce whether the cell producing the curtain is of the appropriate level. A classification as to whether it is a point curtain is made later, due to Lemma 59. Low level curtains are not caught, despite their possible classification as a point curtain, due to discussion later.

Code fragments 28 and 29 may further be referred to as "regulation lifting" for the context of Partial CAD and full CAD respectively. In both contexts for CAD (that is QE or full CAD), we gather all curtains in a container. Hence we move forward to delineate the decomposition technology with modifications to accommodate `QuantifierElimination`. The statement of correctness of the underlying methodologies lies with [56].

Algorithm 30 begins by gathering the set of all top level projection polynomials not used as pivot from P. We perform standard Lazard projection on this set to generate a set of level $n-1$ polynomials. We now gather all non point curtains from the level $n-1$ curtains found in regulation lifting, where the classification is not frustrated by missing

Fragment 29 Full CAD lifting loop in the context of "lifting failures"

```
 1: curtains ← an empty Array
 2: repeat
 3:     cell ← pop( cad, 1 )                          ▷ Any index of cad will do
 4:     if cell ↦ level = n then
 5:         push( leaves, cell )
 6:     else
 7:         try
 8:             CCHILD( cell, cad, bases, vars, n, open, constraints, bounds,
                    curtainCheck = true )
 9:         catch "Lazard curtain":
10:             Let E be the exception caught (about the curtain on cell)
11:             if cell ↦ level = n − 1 then
12:                 Append the list [cell, E] to curtains
13:                 next
14:             else
15:                 ERROR — reraise the exception E
16:             end if
17:         end try
18:     end if
19: until |cad| = 0
```

cells due to Partial CAD, because this is in the context of full CAD post regulation lifting. These non point curtains must undergo further decomposition by the technology from [56], and Algorithm 31 is the recursive part of this. Calling the recursive function returns a container cad_{out} of new level $n-1$ cells from the decomposition of curtains, and an `Array` of "modified" level 1 cells. Usage of the recursive function merges in new geometry via the decomposition of the non point curtains. This may modify existing level 1 cells, which destroys the integrity of the cylinders below those cells. Hence we remove all existing level n cells in the cylinder of such "modified" level 1 cells, before completely relifting the level 1 cells to level n. Finally, we must lift the original level $n-1$ curtain cells (including the point curtains), and the new level $n-1$ cells yielded from the recursive decomposition. All new lifting via CCHILD done in this context is without checking for curtains, because the decomposition is the part that "fixes" the curtains.

Algorithm 31 forms the recursive part of Algorithm 3 from [56], and interprets the pseudocode further in terms of `QuantifierElimination` specific functions and data structures. In particular, because a CAD already exists, but Algorithm 3 of [56] merely creates a CAD with respect to the curtains, decomposeCurtainCellsInner must *merge* in new cells only from new projection polynomials arising from the process of [56]. This is done via incremental technology delineated in Section 5.2, in particular Algorithm 51. Algorithm 31 recurses from level $n-1$ down towards level 1, and returns two containers of cells — a container of new level $n-1$ cells yielded from curtain decomposition, and an `Array` of level 1 cells that are "modified" as a result of merging in geometry at level 1.

These algorithms are specific to "stock" CAD, and so are relevant to CylindricalAlgebraicDecompose. Algorithm 30 is the routine that should be called from CylindricalAlgebraicDecompose when only level $n-1$ curtains were found and gathered in a container curtains (Code Fragment 29). Various remarks on the algorithms, including nuances contrasting to those from the original presentation of the algorithm follow:

1. cad_{in} and cad_{out} are analogous to (I'', S'') and (I', S') respectively from [56]. Meanwhile, "curtains" is analogous to C.

2. One notes that we do not attempt to classify low level curtains as unobstructive point curtains within Algorithm 30, because they are not stored as part of the methodology of Code Fragment 29. This is a conscious decision, because we want to classify all level $n-1$ point curtains with confidence immediately as of Algorithm 30. Otherwise we would fall into a cycle of lifting low level point curtains to attempt to achieve confidence to classify higher level curtains as point curtains, which can now be lifted, etc. This forms an extremely cyclic and convoluted process, especially without propagation of truth values, however the equivalent algorithm for QE somewhat attempts one iteration of this process.

3. Because we did not differentiate between point curtains and non point curtains at the point of stack construction in regular lifting (Code Fragment 27), we do so here amongst the level $n-1$ curtain cells, which are all that can be recovered from, and actually all that remains in the `Array` "curtains" as a result of usage of Code Fragment 29. We use the criteria suggested by Theorem 56 on line 4 to identify the point curtains, using the characterisation of 1-cell neighbours as a "neighbourhood". In particular, in this context, it is unnecessary to use the full criteria implied by Lemma 59, because we will not be impeded by lack of nearby neighbour cells:

   - Low level curtains would have had us erroring out before this point, if they occurred,
   - Truth value propagation removing subtrees of the CAD tree is irrelevant in this context,
   - Regulation lifting has been completed to be in Algorithm 30.

   In total we certainly have every level $n-1$ cell available such that usage of Algorithm 24 returns a complete set. Line 4 selects the cells that get passed to Algorithm 31. Those that are not selected in this way are some of those processed in regular stack construction later in the loop at line 18, and hence point curtains are lifted via regular stack construction.

4. The curtain decomposition works to potentially provide extra level $n-1$ cells to decompose the curtain sets from the gathered non point curtain cells. This is via recursion on lvl, where cad_{out} provides cells of level lvl, hence the top level call with $\text{lvl} = n-1$ provides level $n-1$ cells. n in the presentation in [56] should

Algorithm 30 Curtain cell decomposition for full CAD

Input: P a `projection` object for the CAD, rootCell the root `CADCell` for the CAD, vars an `Array` of variables for the CAD, n number of variables for the CAD, leaves a container for all leaf (level n) cells for the CAD, curtains an `Array` of all level $n-1$ curtain cells for the CAD, open a boolean flag dictating whether we are building an Open CAD, constraints an `Array` of sets of constraints per level deduced from top level lifting constraints, bounds an `Array` of lists of lower and upper bounds per level deduced from top level lifting constraints

Output: No meaningful return, but adds all leaf (level n) cells formed by decomposition of curtains, and those formed by regular lifting on the cells with curtains to leaves

1: **procedure** DECOMPOSECURTAINCELLSCAD(P, rootCell, vars, n, leaves, curtains, open, constraints, bounds)
2: $B_A \leftarrow P \mapsto \text{inequalities}[-1] \cup P \mapsto \text{equations}[-1]$ ▷ B_A all projection polynomials not used as pivot at level n
3: $P_A \leftarrow \text{lazardProjection}(\ B_A, \emptyset, x\ )$
4: NonPCs $\leftarrow$ an `Array` of all non point curtains from curtains, using the criteria on neighbours and Algorithm 26
5: $(\ \text{cad}_{\text{out}}, \text{cad}_m\ ) \leftarrow$ decomposeCurtainCellsInnerCAD(P, P_A, rootCell, vars, n, NonPCs, n, open, constraints, bounds) ▷ Algorithm 31
6: Remove all level n cells from the subtrees of each cell in cad_m from the container leaves
7: **while** $|\text{cad}_m| > 0$ **do** ▷ Regulation lifting on the level 1 modified cells to level n
8: cell $\leftarrow$ pop(cad_m, 1) ▷ Any index of cad_m will do
9: **if** cell $\mapsto$ level $= n$ **then**
10: push(leaves, cell)
11: **else**
12: CCHILD(cell, cad_m, bases, vars, n, open, constraints, bounds, curtainCheck = *false*)
13: **end if**
14: **end while**
15: **for** c **in** cad_{out} **do** ▷ Lift all level $n-1$ cells yielded from the inner decomposition
16: CCHILD(c, leaves, bases, vars, n, open, constraints, bounds, curtainCheck = *false*)
17: **end for**
18: **for** c **in** curtains **do** ▷ Lift all original level $n-1$ curtains
19: CCHILD(c, leaves, bases, vars, n, open, constraints, bounds, curtainCheck = *false*)
20: **end for**
21: **end procedure**

Algorithm 31 Recursive curtain cell decomposition for full CAD

Input: P a `projection` object for the CAD, A a set of polynomials in $\mathbb{R}[x_1, \ldots, x_{\text{lvl}}]$ not necessarily a square-free basis, rootCell the root `CADCell` for the CAD, vars an `Array` of variables for the CAD, n number of variables for the CAD, curtains an `Array` of all level $n-1$ curtains for the CAD, lvl the current CAD level to act upon, open a boolean flag dictating whether we are building an Open CAD, bounds an `Array` of bounds per level deduced from lifting constraints

Output: A container cad_{out} containing all cells created at level lvl via curtain decomposition, and cad_m a container of all level 1 cells modified as a result of merging of new geometry at level 1

1: **procedure** DECOMPOSECURTAINCELLSINNERCAD(P, A, rootCell, vars, n, curtains, lvl, open, bounds)
2: $x \leftarrow$ vars[lvl] ▷ $x = x_{\text{lvl}} \in \{x_1, \ldots, x_{n-1}\}$
3: $\text{cad}_{\text{out}} \leftarrow$ an empty container
4: **if** lvl $= 1$ **then**
5: (B_A, T) $\leftarrow$ univariateBasisAtLazard(A, rootCell)
6: cons $\leftarrow$ constraints[1]
7: $B_A \leftarrow B_A \setminus P \mapsto$ inequalities[1] ▷ Take set difference with all projection polynomials used at level 1 previously in the CAD
8: **if** $|B_A| > 0$ **then**
9: **for** c **in** curtains **do**
10: $R \leftarrow$ getOneCellNeighbours(c)
11: (lb, ub) $\leftarrow$ bounds[1][1], bounds[1][2] ▷ Lower and upper bounds from lifting constraints at level 1
12: $c_\alpha \leftarrow$ the 1st coordinate of the full sample point of c
13: **for** r **in** R **do** ▷ "isolate real roots *between the neighbours of* c"
14: $p \leftarrow$ the 1st coordinate of the full sample point of r
15: **if** $p < c_\alpha$ **then** ▷ See Note 14
16: lb $\leftarrow \max($ p, lb $)$
17: **elseif** $p > c_\alpha$ **then**
18: ub $\leftarrow \min($ p, ub $)$
19: **end if**
20: **end for**
21: incrementalCADMerge(rootCell, B_A, T, x, 1, open, cons $\cup \{\text{lb} \leq x, x \leq \text{ub}\}$, [lb, ub], cad_{out}) ▷ Merge in new cells beneath the root cell corresponding to new roots from new polynomials, adding those new cells to cad_{out}
22: **end for**
23: **end if**
24: $\text{cad}_m \leftarrow$ a container of all level 1 cells (children of the root cell) with bounds modified as a result of calling incrementalCADMerge

Algorithm 31 Recursive curtain cell decomposition for full CAD, Part 2

```
25:     else
26:         ( B_A, cont_A ) ← CADMakeBasis( A, x )
27:         cons ← constraints[lvl]
28:         P_A ← lazardProjection( B_A, cont_A, x )
29:         ( cad_in, modified ) ← decomposeCurtainCellsInner( P, P_A, rootCell, vars,
                n, curtains, lvl − 1, open )
30:         for a in cad_in do
31:             Let α be the full sample point of a
32:             ( f*, T ) ← univariateBasisAtLazard( B_A, a )
33:             Create a CADCell below a with bounds corresponding to bounds[lvl],
                    and add it to cad_out                    ▷ Certainly a (local) sector
34:             if |f*| > 0 then
35:                 for c in curtains do
36:                     R ← getOneCellNeighbours( c )
37:                     ( lb, ub ) ← bounds[lvl][1], bounds[lvl][2]    ▷ Lower and upper
                            bounds from lifting constraints at this level
38:                     c_α ← the lvlth coordinate of the full sample point of c
39:                     for r in R do  ▷ "isolate real roots between the neighbours of c"
40:                         p ← the lvlth coordinate of the full sample point of r
41:                         if p < c_α then                                  ▷ See Note 14
42:                             lb ← max( p, lb )
43:                         elseif p > c_α then
44:                             ub ← min( p, ub )
45:                         end if
46:                     end for
47:                     incrementalCADMerge( a, f*, T, x, lvl, open,
                            cons ∪ {lb ≤ x, x ≤ ub}, [lb, ub], cad_out )▷ Merge in new cells
                            beneath a corresponding to new roots from new
                            polynomials, adding those new cells to cad_out
48:                 end for
49:             end if
50:         end for
51:     end if
52:     return cad_out, modified
53: end procedure
```

be seen as equivalent to lvl in Algorithm 31, instead of n as in the fixed quantity meaning "number of variables for the CAD".

5. The value of lvl to pass at the top level of Algorithm 31 is $n-1$, and not CAD's true "n", not least because examination of the nth coordinate of a level $n-1$ curtain cell would be ill defined in the loop over the curtain cells. Therefore the cells received in cad_{out} from the top level of Algorithm 31 are all level $n-1$, and it remains to lift them to level n (line 15, Algorithm 30). This lifting happens in the standard way, in particular including roots from the level n pivot set (which certainly exists). Hence we can use CCHILD passing the `projection` object P, instead of using the set B_A associated with Algorithm 30, because this excludes that pivot set. Because the child cells created as a result of this lifting are automatically level n, they can be directly added to leaves via CCHILD.

6. It also still remains to lift the (potentially modified) cells that were originally found to have non point curtains. Their child cells are automatically level n cells, and as such can be added directly to leaves (line 18, Algorithm 30).

7. A set difference is taken at level 1 (line 7, Algorithm 31) such that we do not knowingly attempt to construct geometry around roots of projection polynomials that have already been lifted around in standard lifting at the root cell. These set differences are all on fully factored (likely) canonical bases, and as such we certainly remove the possibility of attempting to merge in CAD cells around existing real algebraic numbers at level 1 (Remark 34). This is a specific optimisation in the context of modifying an existing CAD via incrementalCADMerge, in contrast to the original presentation which provides a CAD on the set B_A from Algorithm 30. This only occurs at level 1, because at other levels we act upon new cells which have no original child cells beneath, and the sets of lifting polynomials with which to construct geometry with should certainly be those from the projection on B_A. Line 7 is an optimisation and not to do with correctness — if passed polynomials that were already used in lifting, incrementalCADMerge may invest time in deducing that certain root descriptions had already been used to construct cells.

8. If the aforementioned set difference yields the empty set, then there are certainly no roots to build around at level 1, and the rest of the recursive process does nothing, due to production of no new cells at any time. The case studies of Section 7.2 suggest that the set difference producing the empty set is a common case when curtain decomposition is relevant. In this case, we end up lifting the curtain cells normally despite their non zero Lazard valuations on the top level pivot set. This set difference is specific to the presentation here, due to intention to merge geometry into an existing CAD. The set difference is only relevant at level 1, because at other levels we act upon new cells (a **in** cad_{in}), so the comment about existing geometry is irrelevant.

9. Lines 8 and 34 of Algorithm 31 guard against the pointless expense of tree traversal in order to find all neighbour cells of each curtain cell c when there is no chance

of extra geometry being built anyway, due to a certain lack of new real roots to build around. In practice, programmatically, this is replaced with a similarly cheap check for whether there exists a polynomial in f^* that has any real roots. Else, we needn't take the expense of traversal.

10. Usage of incrementalCADMerge (Algorithm 51) is appropriate considering the intention to modify an existing CAD formed from regular projection & lifting. This is in contrast with the intention of Algorithm 3 of [56] to purely build a CAD of the curtains. Here, the cells that would be created in a vacuum there are merged into the existing CAD, at least at level 1 beneath the root cell, where at least one cell will certainly exist. At other levels, we act upon a new unevaluated cell a created in the last recursion on Algorithm 31, and hence a has no child cells as of yet. But usage of CCHILD on a requires a whole `projection` object, and we merely have one bespoke set to use, and hence usage of incrementalCADMerge is entirely out of convenience, as it takes a set. Hence one creates one cell below a in preparation on line 33 of Algorithm 31, and the new `CADCell`s to create are made as a result of merging, as opposed to the instantaneous production of all of them as would happen in CCHILD. This is the first example of where purely incremental CAD technology gets used not strictly in an overall incremental context. With respect to each new cell a to act upon at line 30, the creation of entirely new geometry below a actually occurs completely incrementally.

11. The recursive curtain decomposition process (Algorithm 31) begins by merging new roots around existing cells at level 1. This may replace previous upper bounds for cells and hence unevaluate them by resetting their sample point and in principle removing the whole CAD subtree beneath. The unevaluation of such cells in terms of removal of their child cells is handled by incrementalCADMerge, but Algorithm 31 needs to handle the removal of the leaves in the cylinder of the modified cells at line 6. This is due to the fact we can no longer be confident of the integrity of the geometry constructed below these cells as an element of their full sample point has changed for each cell. None of these algorithms feature a full tree traversal in order to rededuce the container of all leaf cells, leaves, so we must remove each of the level n cells in the cylinder of any of the modified cells at level 1 manually (line 6, Algorithm 30). Each modified cell undergoes regulation lifting on line 7 such that we can eventually replenish every required leaf cell.

12. There is no opportunity for avoidance of lifting failures at any time in the two algorithms, because the only type we can possibly recover from in full CAD are curtains, which are the entire focus of the process anyway.

13. No polynomials from the results of the projection process attributed to the recursion are added to the original `projection` object P. The decomposition process here is a bespoke correction of the CAD tree specifically on level $n-1$ curtains, and does not reflect the standard lifting process to use under equational constraints. Addition of these projection polynomials would nullify the use of equational constraints and their use in (semi-)restricted projection operators for

any future instances of lifting (for example in incrementality, or even the last loops of Algorithm 30).

14. The arguments "constraints" and "bounds" reflect the need to oblige any potential bounds for each x_{lvl} deduced from lifting constraints passed from CylindricalAlgebraicDecompose (Section 3.6). On line 33 of Algorithm 31, we ensure that the first cell we create below the unevaluated cell a has bounds representing as much of the real line as makes sense taking into account the bounds given. These bounds will be $-\infty$ and ∞ if no lifting constraints were originally passed for x_{lvl}. On line 11, we begin with bounds deduced from lifting constraints before narrowing down these bounds via the sample points from the neighbour cells of the curtain cell. In fact, the sample points from the neighbour cells should be within the bounds deduced by the lifting constraints anyway, due to their initial construction in standard lifting that also took into account these bounds. This forms the deduced bounds $[\text{lb}, \text{ub}]$. Further, we take the union of the deduced constraints $\{\text{lb} \leq x, x \leq \text{ub}\}$ with the appropriate set of lifting constraints "cons" from the variable "constraints" from the top level. We pass this set through to incrementalCADMerge with the deduced bounds such that we completely ensure that we satisfy any relevant lifting constraints with the produced root descriptions from isolation of polynomials in f^*, because the deduced bounds are implicitly judged to form a closed interval, but any of the original lifting constraints may be strict. In any case, this makes use of the "lifting constraints" feature discussed in Section 3.6. The *deduced* constraints narrowed down from the greatest bounds via the sample points of neighbours are not strictly lifting constraints in the sense of Definition 43, but the semantics of their usage entirely coincides with usage of lifting constraints, so they can and should be combined with any original lifting constraints in the relevant variable.

15. Despite the obvious presence of equational constraints to even be processing curtains (that are entirely a result of ECs), the boolean flag open = *true* can be passed to incrementalCADMerge to force merging of an open CAD, as is the freedom of input arguments to CylindricalAlgebraicDecompose. One notes that full CADs for applications in robotics often feature equations, but in reality may only be interested in construction of open space [29].

16. Collections of cells to lift by "regulation lifting" (minus checking for non zero Lazard valuations) within Algorithm 30 are, in order of each subsequent instance of lifting in Algorithm 30:

    - Level 1 cells with their upper bounds replaced by incrementalCADMerge, and then hence their child cells (line 7).
    - New level $n-1$ cells produced by the recursive curtain decomposition (line 15).
    - The original level $n-1$ cells identified as non point curtains (line 18).

    Omission of checking for curtains of any level in lifting the new cylinders of the level 1 modified cells is allowable, because their new cylinders can only be finer

than their previous cylinders were, and the recursive procedure has "corrected" any curtains.

Algorithms 32 and 34 are respective adaptations of algorithms 30 and 31 that cater to the case for QE by Partial CAD. As such Algorithm 32 is the routine to call from any CAD routine performing QE (such as QEPCADL or PartialCylindricalAlgebraicDecompose) when we want to recover from at least one level $n-1$ curtain which cannot be ignored by propagation of truth values. The suffix "Partial" in the names of Algorithms 32 and 34 refer to "Partial CAD" i.e. CAD for QE, in contrast to the suffix "CAD" for Algorithms 30 and 31, which are for "full" or "stock" CAD.

Algorithm 32 begins by classifying whatever level $n-1$ point curtains amongst the cells in problemCells it can, using the confidence criteria implied by Lemma 59. Point curtains are unobstructive, and can be lifted normally, so we remove them from problemCells and lift them first in hope of finding meaningful truth values via regulation lifting. This forms one iteration of lifting point curtains, and it is sufficient to omit checking for curtains (that would trivially reidentify the curtains and add them to problemCells again) because only level $n-1$ cells are lifted here. One could iterate the process of classifying point curtains (in particular, including low level ones) with confidence and regulating lifting them, although this presentation of the algorithm does not do so (Open Problem 62). Regardless, as a result of this process the number of curtains in the container problemCells can only decrease or stay the same. If problemCells is empty as a result of the aforementioned regulating lifting due to propagation of truth values we are done (QE has been deduced as a result). Otherwise, if we have no level $n-1$ non point curtains left, we cannot attempt any recovery from curtains at all. Otherwise, we can enter recursive curtain decomposition in the same manner as that for full CAD, with largely the only modification being that this recursion attempts to propagate truth values from appropriate cells to minimise the amount of decomposition to occur. Much as the case for full CAD, we receive a container of level 1 cells that have been modified, a container of new level $n-1$ cells from the decomposition, and we must also lift all the original curtains. The level 1 cells must be reevaluated due to their modification, and then we regulation lift amongst all these cells (and we can oblige cell selection strategy (Section 3.9) in lifting amongst this collection of cells, as per any instance of regulation lifting).

Unlike the case for full CAD, the level $n-1$ curtain cells may coexist with curtains of any other level in the container problemCells, and the intention is that further usage of Partial CAD may remove any other curtains that cannot be dealt with from the CAD tree to resolve QE. Further remarks on nuances specific to the case for QE follow:

1. Once again we select curtains of level $n-1$ for the decomposition, however this time the container to examine, problemCells is not uniformly of level $n-1$ curtain cells, or even curtain cells at all, so in practice one must examine the reason for stack construction failure (E) and the level of the cell ($c \mapsto$ level) whenever iterating over problemCells.

2. Ideally we would immediately identify the point curtains amongst all level $n-1$

Algorithm 32 Curtain cell decomposition for Partial CAD in Quantifier Elimination

Input: P a `projection` object for the CAD, rootCell the root cell for the CAD, quants an `Array` of the quantifiers $Q_{n-m+1}, \ldots, Q_n$, vars an `Array` of the variables for the CAD, m the number of quantifiers, n number of variables for the CAD, leaves a container for leaf `CADCell`s, problemCells a container for all `CADCell`s where stack construction has failed, constraints an `Array` storing sets of constraints per level deduced from lifting constraints at the top level, bounds an `Array` containing bounds per level deduced from lifting constraints at the top level

Output: No meaningful return, but adds cells formed by lifting or curtain decomposition to the canonical container amongst cad, leaves, or problemCells, and possibly removes various cells from those containers based on propagation of truth values from lifting with Partial CAD. Potentially implicitly deduces QE.

1: **procedure** DECOMPOSECURTAINCELLSPARTIAL(P, rootCell, quants, vars, m, n, leaves, problemCells, constraints, bounds)
2: Eject all certain level $n-1$ point curtains from problemCells, using Lemma 59 and Algorithm 33
3: cad $\leftarrow$ a container of all level $n-1$ point curtains from problemCells
4: Code Fragment 28, applied to cad and the existing container leaves, *without* checking for curtains in CCHILD
5: **if** All $[c, E]$ in problemCells are such that E is not a "Lazard curtain" error **or** $c \mapsto$ level $< n-1$ **then** ▷ All curtains lifted or otherwise removed by propagation, can't do more
6: **return** ▷ If problemCells empty, QE has been deduced, if not then we cannot recover and the calling function errors out accordingly
7: **end if**
8: curtains $\leftarrow$ an `Array` of all remaining level $n-1$ (non point) curtains from problemCells
9: $B_A \leftarrow P \mapsto$ inequalities$[-1] \cup P \mapsto$ equations$[-1]$ ▷ B_A all projection polynomials not used as pivot at level n
10: $P_A \leftarrow$ lazardProjection(B_A, $\emptyset$, x)
11: (cad, modified) $\leftarrow$ decomposeCurtainCellsPartialInner(P, P_A, rootCell, quants, vars, m, n, curtains, leaves, problemCells, n, constraints, bounds)
12: Remove all curtains from problemCells and add such curtain cells to cad
13: **for** cell **in** modified **do** ▷ For all level 1 cells that had modified bounds as a result of decomposeCurtainCellsPartialInner
14: Remove all cells in the CAD subtree beneath cell from any of the containers cad, leaves, and problemCells
15: Reevaluate the truth value of cell, adding it to cad if it has an indeterminate truth value, otherwise leaves. If the evaluation fails, add it to problemCells as a list with its exception.
16: **end for**
17: Code Fragment 28 on cad and leaves to regulation lift the cells from cad, *without* checking for curtains in CCHILD
18: **return**
19: **end procedure**

Algorithm 33 Number of times a `CADCell` is a local extremity

Input: c, a level $0 \leq k \leq n$ `CADCell` (but most likely level $n-1$)
Output: A non negative integer between 0 and k representing the number of times c is a local extremity at any level between 1 and k (e.g. the number of times c is in the cylinder for a cell representing a left-most or right-most sector), allowing us to use the bound for neighbour cells from Lemma 59

```
1: procedure numTimesExtremalCell( c )
2:     if c ↦ level = 0 then          ▷ Really only here as the base case for recursion
3:         return 0
4:     else
5:         p ← c ↦ parent
6:         Let i = c ↦ local_index
7:         if i = 1 or p ↦ children[−1] ↦ local_index = i then
8:             return 1+ numTimesExtremalCell( p )
9:         else
10:            return 0
11:        end if
12:    end if
13: end procedure
```

curtains in problemCells such that we treat them differently to the non point curtains, in the same manner as that of the full CAD case (line 4 of Algorithm 30). This identification can be done using the criteria implied by Theorem 56 to look for non zero valuations on neighbouring cells obtained by usage of Algorithm 24 as the characterisation of a "neighbourhood" of a cell c. However, as per the discussion preceding Lemma 59, the methodology of Partial CAD may frustrate our ability to use this criteria with confidence:

- We do not always uniformly lift to level n cells everywhere in Partial CAD — receiving any determinate truth value on a cell of any level is enough to imply we do not attempt stack construction on that cell.
- We may not even attempt stack construction on a cell c with an indeterminate truth value if truth value propagation from another cell with a meaningful truth value has resulted in removal of the CAD subtree including c.
- Amongst cells with indeterminate truth values that we intend to lift, we could fail to lift, e.g. due to curtains, in particular those of level less than $n-1$.

The above points mean that neighbour cells of a level $n-1$ curtain cell to examine may not all exist in the CAD in terms of the canonicalization implied by Algorithm 24, even having terminated regulation lifting at the top level, which is certainly the case if one is using Algorithm 32. In contrast, for the full CAD case we would not lack any level $n-1$ cells when in the equivalent procedure.

Algorithm 34 Recursive curtain cell decomposition for Partial CAD in Quantifier Elimination

Input: P a `projection` object for the CAD, A a set of polynomials in $\mathbb{R}[x_1, \ldots, x_{\text{lvl}}]$, rootCell the root cell for the CAD, vars an `Array` containing the variables $x_1, \ldots, x_n$, n the number of variables for the CAD, curtains an `Array` containing all level $n-1$ curtain cells, lvl the CAD level for computation, bounds an `Array` containing bounds per level deduced from lifting constraints at the top level

Output: A container cad_{out} containing all cells created at level lvl via curtain decomposition, and a container modified containing all cells modified at level 1 as a result of merging of new geometry

1: **procedure** DECOMPOSECURTAINCELLSINNERPARTIAL(P, A, rootCell, vars, n, curtains, lvl, bounds)
2: $x \leftarrow$ vars[lvl] ▷ $x = x_{\text{lvl}} \in \{x_1, \ldots, x_{n-1}\}$
3: $\text{cad}_{\text{out}} \leftarrow$ an empty container
4: **if** lvl $= 1$ **then**
5: (B_A, T) $\leftarrow$ univariateBasisAtLazard(A, rootCell)
6: cons $\leftarrow$ constraints[1]
7: $B_A \leftarrow B_A \setminus P \mapsto$ inequalities[1] ▷ Take set difference with all projection polynomials used at level 1 previously in the CAD
8: **if** $|B_A| > 0$ **then**
9: **for** c **in** curtains **do**
10: $N \leftarrow$ getOneCellNeighbours(c)
11: (lb, ub) $\leftarrow$ bounds[1][1], bounds[1][2]
12: $c_\alpha \leftarrow$ the 1st coordinate of the full sample point of c
13: **for** c_N **in** N **do** ▷ "isolate real roots *between the neighbours of* c"
14: $p \leftarrow$ the lvlth coordinate of the full sample point of c_N
15: **if** $p < c_\alpha$ **then**
16: lb $\leftarrow$ max(p, lb)
17: **elseif** $p > c_\alpha$ **then**
18: ub $\leftarrow$ min(p, ub)
19: **end if**
20: **end for**
21: incrementalCADMerge(rootCell, B_A, T, x, lvl, *false*, cons $\cup \{\text{lb} \leq x, x \leq \text{ub}\}$, [lb, ub], cad_{out}) ▷ Merge in new cells beneath the root cell corresponding to new roots from new polynomials
22: **end for**
23: modified $\leftarrow$ an `Array` of all level 1 cells (children of the root cell) with bounds modified as a result of calling incrementalCADMerge
24: **end if**
25: PRPTV(rootCell, quants, m, n, cad_{out}, leaves, problemCells)

Algorithm 34 Recursive curtain cell decomposition for Partial CAD in Quantifier Elimination, Part 2

```
26:     else
27:         ( B_A, cont_A ) ← CADMakeBasis( A, x )
28:         cons ← constraints[lvl]
29:         P_A ← lazardProjection( B_A, cont_A, x )
30:         ( cad_in, modified ) ← decomposeCurtainCellsInner( P, P_A, rootCell, vars,
                n, curtains, lvl − 1, open )
31:         for a in cad_in do
32:             Let α be the full sample point of a
33:             ( f*, T ) ← univariateBasisAtLazard( B_A, a )
34:             Create a CADCell below a with bounds corresponding to bounds[lvl]
                    and add it to cad_out                    ▷ Certainly a (local) sector
35:             if |f*| > 0 then
36:                 for c in curtains do
37:                     N ← getOneCellNeighbours( c )
38:                     ( lb, ub ) ← bounds[lvl][1], bounds[lvl][2]
39:                     c_α ← the lvlth coordinate of the full sample point of c
40:                     for c_N in N do▷ "isolate real roots between the neighbours of c"
41:                         p ← the lvlth coordinate of the full sample point of c_N
42:                         if p < c_α then
43:                             lb ← max( p, lb )
44:                         elseif p > c_α then
45:                             ub ← min( p, ub )
46:                         end if
47:                     end for
48:                     incrementalCADMerge( a, f*, T, x, lvl, false,
                            cons ∪ {lb ≤ x, x ≤ ub}, [lb, ub], cad_out )▷ Merge in new cells
                            beneath a corresponding to new roots from new polynomials
49:                 end for
50:             end if
51:             PRPTV( a, quants, m, n, cad_out, leaves, problemCells )
52:         end for
53:     end if
54:     return cad_out, modified
55: end procedure
```

The intention to iterate over neighbour cells to check for non zero Lazard valuations amongst them is ill defined if any are missing, considering we need *all* of them to hold a zero valuation to make the identification for a cell to be a point curtain. This is of course related to the similar problem (in stock CAD or Partial CAD alike) lamented in the discussion following Algorithm 26, where one may not be able to make the classification of a point curtain at the time of stack construction. This is at least because we may not have those neighbour cells available purely in the sense that every cell of a certain level has to be the first to be processed by CCHILD! Having done as much mathematically correct regulation lifting as possible to be in Algorithm 32, we use the bound provided by Lemma 59 for the number of neighbour cells of a level $n-1$ cell ($2(n-1)$), making the bound sharper by also subtracting a neighbour cell for every time an element of the full index of the cell is extremal, via using Algorithm 33. If we find all the neighbour cells and the number of them meets this bound, we know the set of neighbour cells to examine is complete, and hence attempt to identify the point curtains. Having done this on line 3 of Algorithm 32, we lift the point curtains as standard (the regulation lifting at line 4). If performing this lifting leads to the situation where problemCells is now free of level $n-1$ curtains due to propagation of truth values of the newly lifted cells, then no further curtain recovery via the methodology of [56] can even be attempted. If problemCells is empty then QE has been sufficiently deduced, else we are forced to produce an error having returned to the top level. The top level may choose an exception to reraise from problemCells.

3. The only real difference between the case of recursive curtain decomposition between the case for QE and full CAD is the usage of PRPTV on lines 25 and 51 to ensure the output container cad_{out} for building at the next level is of cells with an indeterminate truth value. Usage of PRPTV has us following the spirit of Partial CAD for the CAD being produced on the set B_A, albeit this CAD is being merged into an existing CAD.

4. problemCells, on termination of Algorithm 32 via any **return** statement, is a container of cells where stack construction failed (due to level $< n-1$ curtains or otherwise) which could not be recovered from in the context of Partial CAD, and it being non empty implies Quantifier Elimination could not be deduced.

5. The formulation as a recursive routine where the temporary bases are retained per level means that there is some restriction on cell selection strategy amongst the new cells to act upon from cad_{in} (line 31, Algorithm 34). That is, any metric relating to "level" in cell selection strategy would be superfluous when selecting amongst cells all of the same level in the container cad_{in}. In total, one could adapt this process to an even more bespoke process acting upon arbitrary cells created by this process more in the manner of Code Fragment 28, such that one can, for example, traverse the tree depth-wise as would be the case with usage of cell selection strategy "HL_LI". In this bespoke process, one can retain all the projection bases at once, perhaps in a `projection` object, and an important

adaptation to the process from code fragment 28 is that we must be cognizant of potentially merging in cells at level 1, rather than their regular creation. Another adaptation would be that we would no longer need or want to check for curtains. Currently, the container of cells to choose amongst for "building", cad_{in}, is always uniformly of cells of fixed level lvl at any one recursion, which corresponds to the level lvl.

6. Once again creation of new cells obliges any bounds deduced from lifting constraints at the top level, if they were passed. Lifting constraints can be passed to PartialCylindricalAlgebraicDecompose, and hence may be obliged as a result of this. The logic on their use is essentially the same as the previous case for stock CAD.

7. We instruct incrementalCADMerge to include sections at every level, because presence of curtains is implied by the presence of equational constraints, which excludes the possibility of an open CAD in QE by Partial CAD, because this requires the input formula from the top level to only contain strict inequalities and hence not equations to oblige a request for only open geometry to be built.

8. The recursive function first begins by (potentially) merging in geometry around the root cell, i.e. the child cells of the root cell. In reality, these level 1 cells define a cylinder each, and potentially have child cells. Note that n is strictly more than 1 for these algorithms to even be relevant. When incrementalCADMerge replaces the upper bounds of level 1 cells, such cells are unevaluated by discarding of their sample point, truth value, and child cells. For such cells, we can no longer be confident of the integrity of any of the geometry in the cylinder below them. Hence they are candidates for further stack construction in Partial CAD. These modified cells are identified by addition to the the `Array` "modified". The intention of line 14 of Algorithm 32 is to remove the cells from the subtree of each c in "modified" from the existing canonical containers of `CADCells`, cad, leaves, and problemCells, due to the loss of integrity in such cells. We perform no full tree traversal in curtain decomposition such as the usual case for CAD incrementality to regenerate these containers, so it is necessary to "fix" them. Having added every reevaluated level 1 cell c to an appropriate container, they can be lifted by regulation Partial CAD. In practice it is more likely the level $n-1$ cells created by the curtain decomposition will be lifted first to more likely yield meaningful truth values, considering level is almost always a metric used in cell selection strategy, and this lifting obliges the chosen cell selection strategy via the keyword option `'CellSelectionStrategy'` for CAD in Maple.

9. Item 8 in discussion of the similar algorithms for the full CAD case holds similarly — we take a set difference of polynomials at level 1, which may yield the empty set, meaning the recursive process yields no new cells. The discussion holds essentially the same, considering this nuance is projection and not lifting specific.

10. Despite the presence of various elements of incremental CAD technology, the top level formula for CAD remains static. Hence concepts from CAD incrementality

such as "protection of truth values" are irrelevant (see e.g. discussion point 5 below Algorithm 52). Cells that become unevaluated by usage of incrementalCADMerge lose their truth values, but these are certainly level 1 cells, and we know that the root cell does not hold a *true* or *false* truth value, else Partial CAD would not have us attempting to recover QE via curtain decomposition.

11. There is some opportunity for lifting failure avoidance, via PRPTV taking the argument problemCells on lines 25 and 51 of Algorithm 34. In this way lifting failures arising from evaluation of new cells can be stored, and potentially ignored via the lifting from the recursive decomposition and further regulation lifting in Algorithm 32. However, failures involving root isolation (arising from incrementalCADMerge) are not currently avoided by this methodology, where the "lifting" is too uniform to accommodate as such.

12. Code Fragment 29 appears twice in Algorithm 32. Its first appearance is to attempt to lift point curtains that can be confidently deduced as such via Lemma 59. Ideally, this is sufficient to deduce QE in order to not enter into the recursive projection & lifting process of Algorithm 34. If this was not sufficient, and other level $n-1$ curtains assumed to be non point curtains exist, then the second appearance of Code Fragment 29 as of line 17 is necessary. In both cases, the minor contrast to the presentation of Code Fragment 29 is that we omit checking for any curtain on the cell passed to CCHILD, by passing the keyword option "curtainCheck = *false*". Considering a quantifier free equivalent can't have already been deduced to be in the realms of these algorithms, we know at least one of (sequentially):

    - the lifting of cells deduced to be level $n-1$ point curtains,
    - the recursive curtain decomposition of Algorithm 34,
    - and the lifting of cells found in the container cad as of the second appearance of regulation lifting

    is necessary, but not necessarily sufficient, to deduce QE via propagation of truth values (cells with lifting failures such as low level curtains could remain as of the termination of Algorithm 32). The scope of eligible cells to be lifted from "cad" in the second appearance of the lifting code fragment are:

    - Level 1 cells with their upper bounds replaced by incrementalCADMerge.
    - New level $n-1$ cells produced by the recursive curtain decomposition.
    - The original level $n-1$ cells that could not certainly be identified as point curtains.

    For the same reasons as remark 16 for the case of curtain recovery for full CAD, it is allowable to omit checking for curtains of any level when lifting the new cylinders of level 1 modified cells.

Open Problem 61. *What, if any, distinction & hence potential modifications should occur in the loops on lines 13 and 39 (respectively lines 13 and 40 for the Partial CAD*

case) when the lvl*th element of the full sample point of the curtain neighbour cell* r *coincides with the* lvl*th element of the cell* c*? In other words, do we need an "**elseif** $p = c_\alpha$ **then**", and what does this mean for the deduced bounds* lb *&* ub*, if anything?*

Open Problem 62. *The methodology for dealing with curtain cells in both contexts (full CAD and Partial CAD) currently does not attempt to identify and then lift* all *attainable point curtains (at least regularly, or of any level — Algorithm 32 performs one iteration of lifting of level* $n-1$ *point curtains, helped by propagation of truth values). This process is always frustrated by lack of certainty about point curtains provided by Lemma 59, which means that for example, full CAD always errors out on any low level curtain, such that distinctions are certain on level* $n-1$ *curtain cells later (see note 2 following Algorithm 30).*

One could in theory lift low level point curtains to attempt to achieve confidence in higher level point curtains, and iterate this process. One notes that there is a $\mathcal{O}(n^2)$ *complexity attributed to the tree traversal even to gather neighbours to use the confidence criteria of Lemma 59 on any cell, and we must only omit checking for curtains on identified point curtains, and not further cells in their cylinders, which convolutes the process.*

Can one incorporate a process which guarantees all possible identifiable point curtains (of any level) are lifted before entering any kind of curtain decomposition? Could this still follow the philosophy that the distinction on point curtains should be made as late as possible?

One notes that low level curtains are only relevant in the context of usage of multiple equational constraints in projection. Hence currently only usage of single equational constraints makes CAD with Lazard projection and equational constraints complete, due to Algorithms 30 through 34. Solution of Open Problem 62 somewhat mitigates curtains further in the context of usage of any ECs for Partial CAD, but only further research with further algorithms for multiple equational constraints in the manner of [56] can make CAD in any context fully mathematically complete with multiple equational constraints.

Code Fragment 35 below is the formal methodology for recovering from lifting errors in the context of Partial CAD by performing curtain decomposition. Considering this context, i.e. the context of QE, this code fragment is what would be included late in each instantiation of Partial CAD in `QuantifierElimination`, i.e. PartialCylindricalAlgebraicDecompose, QEPCADL, VTSToCADWhole, CADIncremental, CADDecremental, and modifyCADResult. Of course, the actual variable names may differ slightly, or refer to object properties of `CADData` in evolutionary instances, but the meanings of each variable are entirely equivalent.

In total, the incremental technology offered as part of a greater scheme to enable incrementality for formulae in CAD makes itself useful within the context of modification of a CAD to correct instances of curtains. Section 7.2 includes investigation into examples of curtains and usage of the algorithms in this section for full CAD in order to mitigate their effects in building of a sign invariant CAD. Section 7.4.4 benchmarks QE by Partial CAD with single and multiple equational constraints, to compare both

Fragment 35 Recovery from lifting errors by curtain decomposition if necessary, else exit via exception for Partial CAD

1: **if** |problemCells| > 0 **then** ▷ Recovery from stack construction failure by curtain decomposition
2: **if** There exists $[c, E]$ in problemCells such that $c \mapsto$ level $= n - 1$ **and** E is an exception about a Lazard curtain **then**
3: decomposeCurtainCellsPartial(bases, rootCell, quants, vars, m, n, leaves, problemCells) ▷ Algorithm 32
4: **end if**
5: **if** |problemCells| > 0 **then** ▷ Couldn't fully recover from lifting failures, or potentially even begin to — low level curtains may remain in terms of mathematical errors, otherwise any implementation errors, but not level $n - 1$ curtains
6: **ERROR** by reraising any exception from problemCells, especially one about a low level curtain, if one exists
7: **end if**
8: **end if** ▷ And reaching this point implies recovery was successful or unnecessary

the efficacy in terms of completeness and efficiency between approaches for QE.

Comparison to Implementations using McCallum Projection

For the following implementations, we discuss the handling of nullification occurrences obtained by usage of the McCallum projection with lack of well-orientedness for the polynomials gleaned from input, irregardless of equational constraints. Again, one notes the contrast between a general nullification occurrence and a Lazard curtain explained in Section 3.7.2.

- `ProjectionCAD` offers the ability to produce a warning or an error on production of a curtain. In the case of a warning the computed CAD can be produced irregardless of well-orientedness. `ProjectionCAD` allows a user to continue merely with warnings because a nullification occurrence may only be an *apparent* problem until investigation via delineating polynomials. Being a package not inclusive of a natural implementation of QE, a comparison in terms of "avoidance" or recovery of curtains is irrelevant.

- `QEPCAD B` produces an error about lacking a delineating polynomial upon finding an obstructing nullification occurrence, at which point it is up to the user to restart the computation using the Hong projection $PROJH$, which is guaranteed to be complete, without nullification occurrences.

Via `ProjectionCAD`'s ability to produce a Maple error on a nullification instance, the benchmarking of Section 7.4.3 forces such errors to allow for comparison of the frequency of nullification occurrences in usage of the McCallum projection against Lazard curtains in `QuantifierElimination` for full CADs. The QE benchmarking of Section

7.4.4 is cognizant that `QEPCAD B` may error out on obstructing nullification occurrences with the McCallum projection, and does not attempt to restart with $PROJH$.

3.7.3 Gröbner Bases for Equational Constraints

Gröbner Bases (GBs) are a well known tool in computer algebra. From a set of polynomials in $\mathbb{R}[x_1, \ldots, x_n]$ describing a system of equations, the goal is to produce a basis of polynomials with various properties, delineated in [21, Theorem 16]. Other good overviews of GBs for polynomial ideals are found in [4, 20]. The Gröbner basis for a set of polynomials E generates the same ideal as that for E. We discuss generation of a (reduced) Gröbner basis for the set of ECs E to replace E as this preprocessing.

In terms of the "Tarski" framework, requesting solution to a system of multivariate equations is equivalent to $\bigwedge_{i=1}^{k} f_i = 0$ (as is the relevant top level subformula of (3.2)). Therefore we immediately inherit interest in Gröbner bases via examining the value in preprocessing the equational constraints $f_1, \ldots, f_k$ for a Real Tarski formula. Gröbner bases work over any field, hence it is consistent that we can use the field $\mathbb{R}$ to be able to process ECs from Real Tarski formulae. The native package in Maple implementing procedures to generate Gröbner bases is `Groebner`, with the main procedure of interest Groebner:-Basis. Groebner:-Basis accepts a list or set of polynomials with coefficients as real algebraic numbers represented by RootOfs indexed by interval indices (hence real algebraic numbers in the sense of Definition 30).

Here, we assume $k \geq 0$ equational constraints. If each f_i is represented canonically (as they should be in `QuantifierElimination`), then we can tell that they are distinct, but we cannot immediately tell that the system is over constrained if $k > n$, because there may be non trivial GCDs between the polynomials. For example, $\{x, x^2\}$ is two equational constraints over one variable, but forms a non trivial basis $\{x\}$ via any monomial ordering. This lends an example where our practical number of ECs reduces even over the field of complex numbers. The case for the reals could in theory increase our number of ECs, via $x^2 + y^2 = 0$, which is in practice equivalent to two ECs $x = 0 \wedge y = 0$ over the reals. This is similar to item 3 of the typical examples of ECs in Section 3.7, but one notes `QuantifierElimination` cannot deduce such equivalences. It is only in the reals that we could ever "increase" k, i.e. gain further ECs in this manner — however, the observation $x^2 + y^2 = 0 \equiv x = 0 \wedge y = 0$ is not obtained via GBs.

Forming a flat square-free basis of ECs would be improper, considering the factorisation of any f_i into $g_1 \cdots g_{k_i}$ would imply that each of $g_1, \ldots, g_{k_i}$ belong in the top level conjunction (3.2), but $f_i = 0 \Leftrightarrow \bigvee_{j=1}^{k_i} g_j = 0$, so any factorisation is tied to a disjunction. In particular, Section 3.3 covered that the implementation of equational constraints deals with the backwards implication by multiplying together $g_1 \ldots g_{k_i}$ *without* expansion of the resulting polynomial. The motivation behind redescribing this behaviour of the implementation here is that these polynomials that may attribute some factorisation (that we already know of) enter the Gröbner basis in this form. Hence, we receive an open question:

Open Problem 63. *Can a Gröbner basis use information about (partial) factorisations for the polynomials it receives?*

Usage of Gröbner bases to preprocess ECs in CAD has been of interest in previous works such as [73, 37]. Here, the suggestion of monomial ordering to pass to the Gröbner basis differs significantly, in that it is not the fixed ordering $\texttt{plex}(x_1, \ldots, x_n)$ for $x_1, \ldots, x_n$ the variable ordering to be used in CAD. We note that `tdeg` orderings usually provide a fast Gröbner basis, however usage of a purely `plex` ordering provides a basis with the following property. If $k \geq n$, and the monomial ordering $\texttt{plex}(x_n, \ldots, x_1)$ is used (i.e. a purely lexicographic ordering with $x_1 < x_2 < \cdots < x_n$), and the system of equations has solutions that are zero dimensional, one obtains a"triangular system" of polynomials from the Gröbner basis due to the theory of Gianni & Kalkbrener [32, 42]:

$$\begin{gathered} p_1(x_1), \\ p_{2,1}(x_2, x_1), \ldots, p_{2,k_2}(x_2, x_1), \\ \vdots \\ p_{n,1}(x_n, \ldots, x_1), \ldots, p_{n,k_n}(x_n, \ldots, x_1) \end{gathered} \tag{3.5}$$

where $k_i \geq 1$, $i = 2, \ldots, n$. If $k < n$, or the ECs do not form solutions of zero dimension, then the assumption of obtaining exactly the triangular shape (3.5) is ill founded even when using a fully `plex` ordering, however we will obtain a truncated triangle, or the concatenation of truncated triangles. We discuss the behaviour that we intend to induce from (3.5) in contrast to usage of the monomial ordering $\texttt{plex}(x_1, \ldots, x_n)$, for the moment assuming $k = n$ and the solutions are zero dimensional.

If the output set of equational constraints becomes more "triangular" in the manner of (3.5), then we restrict the scope for early propagation of equational constraints, when multiple equational constraints are used. We attempt to "funnel" the ECs such that they are less numerous amongst the later variables in $x_1, \ldots, x_n$, which are the earlier variables in projection, while also distributing the ECs amongst all levels to ensure as much opportunity for use of ECs in restricted projection as possible. One notes that in (3.5), there are $k_n \geq 1$ level n polynomials, that are certainly of non trivial degree in x_n. Hence there are always $k_n - 1 \geq 0$ propagated ECs at level $n-1$. In contrast, if one is to use the monomial ordering $\texttt{plex}(x_1, \ldots, x_n)$, then the shape of (3.5) is inverted, and *all* ECs are of level n. This does not mean that all the polynomials in (3.5) are necessarily of non trivial degree in x_n (except the analogous univariate polynomial $p_n(x_n)$), but the worst case is that they are, and so a worst case of $1 + \sum k_i$ ECs in x_n, with $\sum k_i \geq n - 1$ propagated ECs. Additionally, in this worst case for $\texttt{plex}(x_1, \ldots, x_n)$, the only scope for usage of ECs beyond projection in x_n are propagated ECs, because all the ECs produced from the GB are pigeonholed at level n. Via properties of resultants, the degree of these ECs is doubling per iteration of projection. Meanwhile, in the case for the new monomial ordering, we obtain a choice between propagated ECs or ECs from the Gröbner basis, but the pivot selection strategy implemented by Algorithm 22 will attempt to minimise degree of the selected pivot, so the scope for choice is useful. At a minimum, because (3.5) donates ECs of monotonically decreasing levels, we ensure we can use as many restricted projection operations as possible. Otherwise, we use the

conjecture that we restrict the scope for propagation of ECs in worst cases, because $k_n < 1 + \sum k_i$, and the number of ECs of non trivial degree in x_n is bounded above by each of these values per the monomial orderings $\texttt{plex}(x_n, \ldots, x_1)$ and $\texttt{plex}(x_1, \ldots, x_n)$ respectively. This sentiment then iterates at lower levels for projection.

We explain the sentiment of attempting to restrict propagation. Nested resultants have the undesirable property that they can lead to spurious roots, which we intend to avoid to optimise the lifting process. At lower levels, propagated ECs are potentially nested resultants of ECs from higher levels. Propagation of equational constraints is always desirable in the sense of providing as many ECs as possible to enable as much restricted projection as possible and hence minimising the projection bases. However, if we can reduce the number of nested resultants owing to ECs then we can attempt to minimise the number of spurious roots. This appears to be relevant only when both propagation of ECs and multiple ECs are enabled (`'UseEquations' = 'multiple'`, `'PropagateECs'` = *true*), but it is even relevant when using just a single EC, because we still intend to restrict the scope for propagation in x_n. Technically, the usage of GBs to receive (3.5) can completely replace propagation of ECs, because the resultants of ECs in x_i can be constructed via linear combinations of already existing ECs in x_{i-1}, hence forming the same ideal, but the implementation does not currently use this fact. To restrict the possibility of nested resultants further, we could generate Gröbner bases at lower levels (with an appropriate truncated monomial ordering) on the appearance of new ECs generated via the resultant rule, but one considers that the degree of ECs created via propagation are doubling per iteration of projection via properties of resultants, and of course the complexity of Gröbner bases is dependent on degree. Once again we note `QuantifierElimination`'s implemented pivot selection strategy attempts to select a pivot of minimal degree at any level, even with a view to minimising the degree for the next level.

Example to Compare Monomial Orderings

As an example of where the new monomial ordering is more effective than that used in previous literature, consider the example

$$\exists a \exists b \; a + b + c = 0 \wedge ab + bc + ac = 0 \wedge abc = 0$$

which is "Cyclic-3" from the QE examples database (Section 7.1). This example is entirely ECs (hence $k = n = 3$), but not *fully* existentially quantified, which would lend itself entirely to solution via GBs and back substitution via Gianni–Kalkbrener from (3.5). Instead, c is unquantified to explore the conditions on c such that the system of equations has solutions. The equations are entirely symmetric with respect to variables, so orderings are superfluous beyond them being valid. Take the variable ordering $[c, b, a]$. Under the monomial ordering $\texttt{plex}(c, b, a)$, the Gröbner basis for the set of ECs $\{a + b + c, ab + bc + ac, abc\}$ is $\{a^3 - 1, a^2 + ab + b^2, a + b + c\}$, and one notes that all polynomials are level 3, i.e. all of non trivial degree in a. This is exactly the worst case from this monomial ordering that we intended to avoid, and there is a maximum amount of propagation at every level, with PartialCylindricalAlgebraicDecompose yielding 11 projection polynomials and 119 leaf cells in total from the QE. In

usage of the monomial ordering $\texttt{plex}(a, b, c)$, we receive $\{c^3 - 1, b^2 + bc + c^2, a + b + c\}$, with one polynomial of each level ($k_i = 1$, $i = 2, 3$, in terms of (3.5)), and so there is no scope for propagation of ECs at any time. As a result there are fewer projection polynomials with a total of 6, and as a corollary, many fewer leaf cells with a total of 25. Section 7.4.3 investigates usage of GBs to reduce the number of leaf cells further (in the case of full CAD), additionally examining this example further.

We discuss some logic on examination of the output GB from processing of ECs. If the Gröbner basis output is $\{1\}$ (or any $\{p\}$ such that $\deg(p) = 0$) then the system of equational constraints $\bigwedge_{i=1}^{k} f_i$ has solutions of dimension $-\infty$, and so is equivalent to *false* (as $1 = 0 \equiv \mathit{false}$). Considering this is a subformula of a top level conjunction, the whole formula is then equivalent to *false*, so the rest of the formula is superfluous to solve the system. In particular the empty projection (no polynomials at every level) is sufficient to imply lifting the single cell $\mathbb{R}^n$ as the CAD. Substitution of any sample point $\alpha \in \mathbb{R}^n$ into Φ will give *false*, and as such the production of the empty projection was sufficient to canonically give the quantifier free equivalent to Φ via CAD. This situation may arise as a result of $k > n$ and the system actually having been over constrained. If the Gröbner basis output is $\emptyset$ (the empty list `[]` in Maple), then the conjunction of equational constraints has solutions of dimension ∞, and so is equivalent to *true*. As this is equivalent to the set of clauses of a top level conjunction (3.2), they can be discarded (or, semantically, $\emptyset$ is used as the set of equational constraints in their stead).

When $k < n$, the aim to create a fully triangular system is generally ill founded. Therefore we needn't use a completely purely lexicographical ordering, which may be needlessly inefficient. We can use "`tdeg` in the first $n - k$ variables" to attempt to introduce some extra efficiency to the Gröbner basis where the generated basis can clearly not yield an exactly triangular system in the sense of (3.5), and we may as well order the first $n - k$ variables within this ordering with respect to some heuristic, because they are irrelevant to the triangular system we wish to coerce amongst the last k variables. Coercing a triangular system towards the later variables is important as they are the first variables to use in projection, and optimising early projection steps has effects on the later ones. Hence we need to keep "`plex` in the reverse of the last k variables". Maple offers two constructs to describe non standard monomial orderings. Matrix orderings allow for the largest scope, but here we just require a "product ordering":

$$\begin{aligned}\texttt{prod}(\,&\texttt{plex}(x_n, \ldots, x_{n-\min(n,k)+1}),\\ &\texttt{tdeg}(H(x_1, \ldots, x_{n-\min(n,k)}))\,)\end{aligned} \tag{3.6}$$

which is the appropriate monomial ordering to pass as a second argument to Groebner:-Basis, where $H(x_1, \ldots, x_{n-\min(n,k)})$ represents usage of any heuristic to sort $x_1, \ldots, x_{n-\min(n,k)}$. For H, `QuantifierElimination` uses the built in function Suggest-VariableOrder from the `Groebner` package in Maple, i.e. Groebner:-SuggestVariableOrder. In passing the set E and the list $[x_1, \ldots, x_{n-\min(n,k)}]$, we obtain a sequence corresponding to the suggested ordering by Maple on $x_1, \ldots, x_{n-\min(n,k)}$ in terms of E. Note that if $k \geq n$ then the above is entirely equivalent to $\texttt{plex}(x_n, \ldots, x_1)$. `plex` appears in the

product first to ensure coercion of the triangular behaviour, such that ties with respect to $\texttt{plex}(x_n, \ldots, x_{n-\min(n,k)+1})$ are broken by $\texttt{tdeg}(H(x_1, \ldots, x_{n-\min(n,k)}))$.

equationalConstraintsToGroebner is the function taking a set of equational constraints E and a fixed variable ordering via `Array` $[x_1, \ldots, x_n]$, generating the Gröbner basis for E with monomial order given by (3.6), or $\texttt{plex}(x_n, \ldots, x_1)$ if $k \geq n$. The previously discussed prescribed logic on usage of the Gröbner basis, i.e. the set of equational constraints after processing is handled by the function handling projection — this can be seen in Algorithms 5 or 50.

Usage of Gröbner bases to preprocess equational constraints is controlled by the keyword option `'UseGroebner'` for the top level functions of `QuantifierElimination`. This option takes a boolean flag to enable or disable their use, and by default it is *true*. This flag appears as arguments to Algorithms 5 or 50. In usage of no equational constraints via the keyword option `'UseEquations'` = `'none'`, the value of the keyword option `'UseGroebner'` is irrelevant, due to the structure of Algorithm 5.

Currently, the ordering of the variables within the monomial ordering is dependent on their ordering for projection, i.e. the CAD variable ordering. For most variable strategies, variable ordering is fixed before (any individual) full projection, apart from the "greedy" variable strategy described in the following subsection, which makes decisions on variables based on intermediate results of projection. It is unclear how to use the described Gröbner basis preprocessing when this variable strategy is used, because this is the only strategy where we do not fix the ordering of all n variables ahead of projection. Hence when this strategy is selected, the value of `'UseGroebner'` is implicitly set to *false* and Gröbner bases are not used (but the user is warned accordingly). Some analysis on the efficacy of Gröbner bases via these monomial orderings with respect to the example sets used in benchmarking appears in Section 7.4.3.

3.8 CAD Variable Strategy

Definition 64 (Variable Strategy/Ordering). *A* variable strategy *is the name of a strategy to use to obtain a* variable ordering *for one input.*

Remark 65. *This means that the variable* ordering *inherited from usage of two distinct* strategies *on the same input may coincide.*

`QuantifierElimination` offers customizable variable strategy whenever CAD is concerned via the keyword option `'VariableStrategy'`, which accepts a symbol corresponding to each of the strategies described in this section, or a list of variables from the example to parse, which is checked for correctness and compatibility in terms of commutativity of any present quantifiers. when QE is relevant. In this way the user can enforce a variable ordering to use in lieu of selection of a particular strategy.

The nature of the projection and lifting steps of CAD has that CAD is very sensitive to variable ordering, which in itself may be guided by the intermediate results of projection. We once again note that variable strategy is limited by the fact that only quantifiers with the same symbol commute, so in reality variable strategies are only

sorting variables within blocks. This includes commutativity of free variables, and so for "stock CAD", there is complete freedom for commutativity.

Success of metrics for what makes a "good CAD" can often be predicted by metrics on what makes a good projection, considering the CAD lifting process essentially completely follows from projection. Such metrics on the projection bases however vary in their expense. For example, some of these metrics require calculating the whole set of projection bases for all the $\mathcal{O}(n!)$ possible variable orderings, and then picking the best such ordering via applying the metric to each set of bases. Note that this $\mathcal{O}(n!)$ is really a worst case, given that only similarly quantified (or unquantified) variables commute, and hence not every permutation of the n variables is permissible. The most simple metric however actually requires no projection, and will provide an ordering immediately from information that can be garnered from the input polynomial set(s). We first define the metrics on projection bases with respect to a fixed (potentially temporary) ordering. For the purposes of these definitions, let $B_1, \dots, B_n$ be *flat* sets of polynomials per canonical CAD level in projection, i.e. B_i is the union of all polynomials in bases at level $1 \leq i \leq n$ in projection with respect to Figure 3-3 or 3-4, i.e. irregardless of any equational constraints appearing at that level.

Definition 66 (sotd). *Let $B = B_1, \dots, B_n$ be the projection bases under some variable ordering $\mathbf{x} = x_1, \dots, x_n$. Dolzmann et al provide the* sum of the degrees *(sotd) metric in [25].*

$$\text{sotd}(B, \mathbf{x}) = \sum_{i=1}^{n} \sum_{f \in B_i} \sigma(f)$$

where if $e = (e_1, \dots, e_n)$, and $f = \sum_{e \in E} a_e x_1^{e_1} \dots x_n^{e_n}$ (i.e. a sum of monomials where each e in E defines the exponents), then

$$\sigma(f) = \sum_{e \in E} \sum_{i=1}^{n} e_i$$

hence in other words sotd is the sum of "total degrees" of each monomial in each polynomial in each projection basis in B.

Definition 67 (ndrr). *Let $B = B_1, \dots, B_n$ be the projection bases under some variable ordering $\mathbf{x} = x_1, \dots, x_n$. Bradford et al provide the* number of distinct real roots *(ndrr) metric in [5].*

$$\text{ndrr}(B, \mathbf{x}) = \sum_{f \in B_n} \textit{number of real roots of } f \textit{ in } x_n \textit{ via real root isolation}$$

ndrr is the only heuristic to consider the real geometry, directly giving information about the number of cells one will inherit at the first stage of lifting to build a CAD of $\mathbb{R}^1$, but requires real root isolation, which is one of the more costly parts of lifting. Meanwhile, the inherent operations to calculate sotd are essentially trivial. However, in order to be used as metrics, one needs to compute multiple full projections in the manner of Algorithm 5 in order to decide on a variable ordering. Usage of sotd & ndrr via computation of all admissible full projection bases are strategies offered by

`QuantifierElimination` via usage of the keyword option `'VariableStrategy'`, e.g. `'VariableStrategy' = 'ndrr'`. [5] additionally discusses ndrr and sotd as metrics to examine the resulting projection sets in order to designate a single equational constraint (i.e. selection of a pivot when a single EC is admissible, there in the context of the McCallum projection).

[25] suggests a slightly less verbose manner of usage of sotd, namely the "greedy" strategy. Here, at level $1 \leq i < n$, one computes all projection bases from amongst the remaining (permissible per quantifiers) $n - i + 1$ variables, and upon usage of the sotd metric to decide on the best resulting bases, *fixes* the variable x_{n-i+1}. Hence there are fewer permutations of variables to inspect, and only one maximum level projection basis is ever produced. The same modification cannot be used with ndrr, because the metric is inapplicable to an incomplete set of projection bases. This strategy is offered in `QuantifierElimination` via `'VariableStrategy' = 'Greedy'`. Any `projection` object associated to the best ordering under any of these metrics is trivially retained for lifting the CAD, and in particular the "greedy" strategy induces a method acting upon such an object. The implementation of the greedy strategy in `QuantifierElimination` follows the methodology of [25] while paying attention to equational constraints in any one variable much in the same way as Algorithm 5.

Remark 68. *The "projection based" metrics (ndrr, sotd, greedy) are in practice informed by usage of equational constraints in projection, with respect to the Lazard projection CAD with ECs in* `QuantifierElimination`*. Further, they are affected by the value of the keyword options* `'UseEquations'` *— usage of anything up to multiple equational constraints, the value of* `'PropagateECs'`*, — propagation of equational constraints, and the value of* `'UseGroebner'` *— usage of Gröbner bases to preprocess the ECs.*

The package `ProjectionCAD` in Maple implementing projection & lifting CADs via the McCallum projection offers a variable strategy (by default) that allows for weighted use of the sotd and ndrr metrics for selection of a set of full projection bases to use. `QuantifierElimination` currently does not offer such an option — this is perhaps further work for development. Both metrics are readily available to use on a set of full projection bases, so the adaptation is not difficult, and the complexity of usage of a weighted strategy remains at generation of $\mathcal{O}(n!)$ full projection bases. The question of the weighting is a matter of investigation and benchmarking, and one notes Sections 7.4.2 and 7.4.3.

The final metric from other research is the understated "Brown heuristic" from [13]. This has negligible complexity implications in comparison to the above. Considering usage of the heuristic essentially comes "for free", it is good news that it seems to quite often pick a very good ordering, as seen in [5].

Definition 69 (Brown Heuristic). *Let A be the set of input polynomials for CAD (as a flat set, i.e. $A \leftarrow A \cup E$, when E is the top level set of equational constraints).*

The "Brown heuristic" from [13] implies that we should eliminate variables (i.e. take the projection of) in order according to the following tiebreakers:

1. *it has a lower overall degree in A:* $\max_{f \in A} \deg_x(f)$,

2. *it has a lower maximum total degree of those terms in the input in which it occurs:* $\max_{f \in A \mid \deg_x(f) > 0} \deg(f)$

3. *there are fewer terms in the input which contain the variable:* $\sum_{f \in A \mid \deg_x(f) > 0} 1$

The Brown heuristic makes no special use or reference to equational constraints in contrast to polynomials from regular constraints. This is not necessarily out of ignorance, but could be by design to cater for usage of projection operators without equational constraints. `QuantifierElimination` offers usage of the Brown heuristic as a strategy for sorting blocks of quantifiers via `'UseEquations' = 'Brown'`. The following Algorithm 36 is a simple modification to the Brown heuristic to take into account a set of equational constraints E alongside the usual set of polynomials from inequalities A.

Algorithm 36 ECHeuristic

Input: x, y, the names of variables of polynomials in A and E, A a set of polynomials associated purely with inequalities from input for CAD, E a set of equational constraints for CAD
Output: *true* if $x < y$ with respect to this heuristic, else *false*

1: **procedure** ECHeuristic(x, y, A, E)
2: $m_x \leftarrow \max(\ \{\deg_x(p) \mid p \in E\}\ )$
3: $m_y \leftarrow \max(\ \{\deg_y(p) \mid p \in E\}\ )$
4: **if** $m_x \leq 0$ **then**
5: $m_x \leftarrow \infty$
6: **end if**
7: **if** $m_y \leq 0$ **then**
8: $m_y \leftarrow \infty$
9: **end if**
10: **if** $m_x < m_y$ **then** ▷ Tiebreaker 1 from Definition 69 applied to E
11: **return** *false*
12: **elseif** $m_y < m_x$ **then**
13: **return** *true*

One notes that one applies each successive tiebreaker from the Brown heuristic in Definition 69 to E before applying the same tiebreaker to A. In particular, the order of examination is E, A, E, A, E, A, and not E, E, E, A, A, A. Another small modification are lines 5 and 8. The intention of these lines is to coerce as many equational constraints, and hence their usage in restricted projection (hence yielding fewer polynomials) as early as possible in projection (recall that projection acts backwards across $x_1, \ldots, x_n$). Therefore, to have trivial degree in a variable x is as bad as to have excessively high degree. This also discourages a variable being designated as x_1 when there exists a valid equational constraint that could restrict a preceding projection step (no projection operations are ever performed on x_1). Usage of the Brown heuristic as a variable strategy reduces to usage of a very similar function within

Algorithm 36 ECHeuristic, Part 2

```
14:     else
15:         if m_x = ∞ then
16:             m_x ← −∞
17:         end if
18:         if m_y = ∞ then
19:             m_y ← −∞
20:         end if
21:         m_x ← max( {deg_x(p) | p ∈ A} )
22:         m_y ← max( {deg_y(p) | p ∈ A} )
23:         if m_x < m_y then              ▷ Tiebreaker 1 from Definition 69 applied to A
24:             return false
25:         elseif m_y < m_x then
26:             return true
27:         else
28:             for S in E, A do▷ Tiebreaker 2 from Definition 69 applied to E, then A
29:                 m_x ← min( {deg(p) | p ∈ S, deg_x(p) > 0} )
30:                 m_y ← min( {deg(p) | p ∈ S, deg_y(p) > 0} )
31:                 if m_x < m_y then
32:                     return false
33:                 elseif m_y < m_x then
34:                     return true
35:                 end if
36:             end for
37:             for S in E, A do▷ Tiebreaker 3 from Definition 69 applied to E, then A
38:                 m_x ← Σ_{p∈S} 𝟙_{deg_x(p)>0}
39:                 m_y ← Σ_{p∈S} 𝟙_{deg_y(p)>0}
40:                 if m_x < m_y then
41:                     return false
42:                 elseif m_y < m_x then
43:                     return true
44:                 end if
45:             end for
46:             return false        ▷ return true only if x < y, but really they are equal
47:         end if
48:     end if
49: end procedure
```

`QuantifierElimination`, with fewer tiebreakers, treating A and E as the same (essentially acting on their union). `QuantifierElimination` offers usage of ECHeuristic to sort permissible variables via passing the option `'VariableStrategy' = 'Tonks'`.

One notes that polynomials in E may have some partial factorisation to them, due to the discussion on gathering ECs via multiplication without expansion in Maple using Algorithm 21 in Section 3.7. Because we want to retain these factorisations to choose pivots in usage of ECs, the intention to compare the degrees of such polynomials from E is well informed in usage of either the Brown heuristic or ECHeuristic as long as we genuinely intend to use those ECs, i.e. `'UseEquations'` is not `'none'`, else each heuristic is being led astray by a non natural multiplication amongst polynomials that will be fully factored in the usual way in the projection process. Clearly, this is because the degree of a product of polynomials is the sum of the degrees of the factors. Then again, this behaviour can also equally be inherited from polynomials in A that are not irreducible, because there is no requirement on the set A or E to be fully factored or even square-free bases. As such, both aforementioned heuristics are best informed when the formulation of input by the user donating the sets A and E is as concise as possible.

[30, 31] investigate usage of Machine Learning (ML) to generate heuristics for variable strategy in CAD. Considering `QuantifierElimination`'s role as a package in Maple, the results of a call to any outward facing procedure should not change from call to call as a result of usage of ML data. In particular this mandation comes as a result of acknowledgement that ML may "learn" from past calls. On the other hand, it is reasonable (and perhaps desirable) for the results of a call to change from release version to release version (in particular improve in some way). No ML heuristics are currently implemented in `QuantifierElimination`, but this is interesting further work.

Considering there exist variable strategies for CAD that are entirely tied to the projection, `QuantifierElimination` mandates the semantics that a variable strategy should return the set of projection bases implied by the variable strategy to lift from (the purpose of CADChooseVarsProjection).

The most popular metric to measure efficacy of CAD output (such as in [25]) is on the number of leaf cells. This was however usually used in work that calculated a full CAD (these works had no direct context of QE). In using partial CAD, usage of "total leaf cells" as a metric may not be useful considering the CAD tree at output may not be complete, as a result of deducing termination early. In particular the presence of cell selection strategy (Section 3.9) for stack construction complicates matters, and individual partial CADs may differ despite arising from the same projection bases. Remark 68 raises the question as to whether `QuantifierElimination` may find more efficacy from the projection based heuristics if we find that usage of equational constraints makes `QuantifierElimination`'s CAD even more sensitive to variable ordering than past implementations benchmarked e.g. in [25]. Comprehensive benchmarking on full CAD, per variable strategy, implementation, usage of equational constraints, Gröbner bases, and propagation of equational constraints in Section 7.4.3. The output of such

benchmarks takes into account low level curtains and other lifting errors, but otherwise measures the number of leaf cells produced, and the time and memory expended in computation.

Considering the projection based orderings may attribute significant expense as well, Section 7.4.2 investigates the time to complete projection and calculate the variable ordering for the same examples in isolation (because for the projection based orderings, these are intrinsically tied). Hence one can begin to examine the cost of projection and lifting separately per strategy.

In total, variable strategy within CAD in `QuantifierElimination` is customizable via the keyword option `'VariableStrategy'`, taking the name of one of the aforementioned strategies to use. It is also overridable by a list of variables which will force an ordering defined by the user, which is checked for validity not least in terms of the commutativity of any quantifiers.

Gröbner Bases with "Heuristic" Variable Orderings

Usage of Gröbner bases on a set of equational constraints E can actually remove polynomials from E. For example, if $E = \{x^2y, y\}$, and x appears nowhere in the set of polynomials from inequalities A, then the Gröbner basis for E under any monomial ordering on x and y is $\{y\}$, because only y need be 0 such that $x^2y = 0 \wedge y = 0$. The "heuristic" variable orderings Brown and ECHeuristic inspect the incoming top level sets A and E before E's preprocessing via Gröbner basis. Both heuristics (in particular ECHeuristic, which pays more attention to those polynomials from E than those from A) can hence be led astray by variables that have the capacity to be "removed" from the polynomial sets after Gröbner bases. Here, either heuristic can be led astray to view x as higher degree than any variable appearing linearly in A, despite the fact that after processing by Gröbner, x is of degree 0 in every polynomial, and hence should be viewed as lower degree than any variable appearing linearly in A.

One could delay usage of variable strategy until after preprocessing of E via Gröbner bases, but the monomial ordering is dependent on the results of variable strategy to generate an ordering, forming a cyclic dependency. `QuantifierElimination` takes the view that ECs are rarely ill defined to cause this behaviour, and so is a problem of good formulation of the problem to pass to QE.

The overall intention of ECHeuristic in conjunction with Gröbner basis preprocessing is that the former attempts to coerce as many equational constraints (ideally in a row) as early as possible in projection to maximise opportunity for propagation of ECs, and the Gröbner basis preprocessing then attempts to restrict the early opportunities for propagation. This may seem like a contradiction, but one notes that the latter step only attempts to make the propagation as efficient as possible — we would prefer to use propagation of ECs where possible, but if we are to use propagation, we would then prefer to produce as few nested resultants as possible.

3.9 Cell Selection Strategy

Cell selection strategy is strictly of interest when the early termination criteria such as from Partial CAD applies. As such it is only implemented and used in the QE relevant functions from `QuantifierElimination`. It is not used in CylindricalAlgebraicDecompose, because the aim is to compute and provide every leaf cell without termination criteria, and as such would be unnecessary overhead. For the relevant functions, `QuantifierElimination` implements four cell selection strategies directly from the "Partial CAD" work [36]. They are:

- `'HL_LI'`
- `'TC_LD_HL_LI'`
- `'TC_LD_HL_GI'`
- `'SR_HL_LI'`

The glossary of abbreviations here is:

HL	"highest level"
LI	"least index"
GI	"greatest index"
TC	"trivial conversion"
LD	"least degree (of the minimal polynomial defining the cell)"
SR	"sectors first"

The names of the strategies can be read as successive tiebreaker conditions, with the underscore as a delimiter between those tiebreakers. Note that ties can always be broken, given that the last tiebreaker condition is always something involving the (full) index of a cell, and these are unique between cells. "Highest level" always appears as a metric in every strategy. This is unsurprising, given it is the metric most likely to lead to early termination for "SMT-like" problems, i.e. those which are quantified similarly or close to similarly.

Cell selection strategy is handled by the function QECADStrategy, which uses the strategy defined by the CAD keyword option `'CellSelectionStrategy'` to return the index of the cell suggested by that strategy. The value of `'CellSelectionStrategy'` is any of the symbols from `'HL_LI'`, `'TC_LD_HL_LI'`, `'TC_LD_HL_GI'`, or `'SR_HL_LI'`. The default value is `'HL_LI'` (the same suggested as default in [36]). The container of evaluated `CADCell`s with indeterminate truth values amenable to further stack construction to choose between in regulation lifting in any context is called "cad". This container must support addition and removal of objects in similar style to one containing non genuine leaf `IQER`s. In `QuantifierElimination`, this is the `QEContainer` object which is a fairly rudimentary object supporting iteration, addition and removal methods, as is also used in VTS (Section 2.3.1). In particular, it is rudimentary in the sense it does not directly reconcile with the chosen strategy — QECADStrategy iterates over the

whole container, retaining the index of the best cell with respect to the strategy and the associated information with respect to the metrics in order to enable comparison. Hence the usage of strategy to extract the "maximal element" is $\mathcal{O}(k)$ where k is the current number of evaluated non meaningful leaf cells, which can of course be exponential at least in the number of variables. The usage of the above metrics can of course be deemed to expend essentially constant work on fairly trivial facts about integers, or at worst looking at degrees of polynomials.

Much as the case for the corresponding container of "work" `IQER`s in Section 2.3.1, usage of a `QEContainer` could be replaced by a more sophisticated data structure which maintains an ordering amongst cells for future stack construction corresponding the selected strategy with respect to one CAD call. Unsurprisingly, experimentation via code profiling shows that usage of CAD strategy is not a bottleneck to the implementation, but optimisation of strategy may be of interest in the future. The suggestion of a heap differs slightly from the case for VTS, because of the methodology of Partial CAD, with propagation of truth values via Algorithm 15 implying the desire to be able to remove arbitrary CAD subtrees of cells from "cad". Heaps are actually essentially equal to the `QEContainer` in this respect, because in both cases removal of an arbitrary non maximal cell is $\mathcal{O}(k)$, with the main expense attributed to "finding" the cell to remove within the structure in both cases. In the case of a heap, we additionally require "heapifying" a subtree of the structure after removal, attributing an additional $\mathcal{O}(\log k)$. Arbitrary removal of elements is irrelevant in the case of fully homogeneous problems (such as `QF_NRA`), where usage of a heap now becomes even more lucrative, but `QuantifierElimination` aims to support general quantifier elimination. However, switching to a heap still appears to be a win for general QE due to the reduction in expense of pairs of addition/"extraction of maximal element" operations to $\mathcal{O}(\log k)$ from $\mathcal{O}(k)$, with a minor complication being the expense of addition of cells that only ever see arbitrary removal rather than their extraction as a maximal element for stack construction, much as the similar case for VTS. Other potential options are AVL or red-black trees, which attribute $\mathcal{O}(\log k)$ expense to all the aforementioned operations including addition, arbitrary removal, and extraction of maximal elements.

The object "cad" never needs to be retained between incremental CAD calls (via the top level algorithms in Section 5.2), so the fact usage of strategy for a `QEContainer` is highly mutable within the context of one CAD construction is not largely to its credit.

Of course, the `QEContainer` may as well remain for the implementation of full CAD (CylindricalAlgebraicDecompose), where cell selection strategy is entirely superfluous, and picking e.g. the last cell from the container for stack construction is entirely sufficient, because we need do this for every non leaf cell.

3.10 Production of Witnesses for QE via CAD

We discuss witnesses in the sense of Definition 22, but now in the context of CAD, i.e. with respect to meaningful leaf `CADCell`s. Witnesses are only relevant for homogeneously quantified problems, hence a meaningful leaf `CADCell` in this context always propagates its truth value all the way to the root.

Witnesses for CAD are easier to produce than for VTS, because every substitution that occurs in lifting is of exactly one real algebraic number. In some sense, the lifting process is akin to back substitution, and hence no back substitution to do to deduce the actual substitutions in lieu of "virtual" substitutions. The full sample point for a `CADCell` (i.e. the concatenation of the local sample point for a `CADCell` with those from its parent, recursively) is precisely the list of witnesses for a meaningful leaf `CADCell` for a fully homogeneously quantified formula. Due to cylindricity, one could even use the cell description for such a cell as a witness, due to that `CADCell`'s truth invariance on Φ — this gives a more comprehensive "proof" considering it gives the whole subset(s) of $\mathbb{R}^n$, but is much less convenient in terms of substitution to obtain the "proof". Each `CADCell` only stores data local in terms of that level, so the witnesses need to be gathered by traversal towards the root cell (collecting the local sample points of each cell hit along the way).

The elements of a sample point for a `CADCell` are all real algebraic numbers. For a sector, we can always find a (simplest) rational number as a local sample point. For sections, the local sample point is the real root of some lifting polynomial p represented as $\text{RootOf}(p, a..b)$ (if p is not linear) in Maple for some $[a, b]$ the root isolation about that real root. Hence evaluation of any formula at such a sample point in Maple may require usage of `evala( Normal( ... ) )` in order to simplify the substitution of algebraic numbers, as can be seen in Figure 3-10.

3.11 Comparison with VTS

The substitution points from one propagation of VTS owe completely to the roots of the polynomials from the `IQER` to propagate on, ignoring boolean operators. As such, the success of substitution of a VTS test point from any one polynomial is predicated entirely on the boolean structure of the formula to substitute into, and more importantly the test point's guard. However, such substitution points may contain infinitesimals, and even other variables, and as such the substitutions themselves are more complicated compared to those that CAD makes, reducing to pseudoremainders and production of other formulae. Indeed, any one substitution in CAD will use a rational number, or at worst a real algebraic number. However the nature of such substitutions arising from polynomials from projection bases means that such points being used may not be meaningful in terms of the boolean structure of Φ, but worse, possibly even real space.

In the worst case, CAD will project using all n variables in $\exists x_1\ x_1 < 0 \wedge x_1 > 0 \wedge f_1(x_1, \ldots, x_n) \wedge \cdots \wedge f_k(x_1, \ldots, x_n)$, despite the trivial falsity of the first two operands of the conjunction (assuming no use of lifting constraints). Meanwhile VTS will deduce the falsity of the formula due to any and all substitutions for x_1 immediately producing *false* for at least one of the first two operands, assuming no ineligibility arising from $f_1, \ldots, f_k$ and noting the linearity of x_1. While the best case for VTS clearly looks good here — substitutions of just 3 test points for x_1 to deduce *false*, the worst case for this example is in fact as many test points as are found for x_1. A related point is that VTS largely ignores free variables — although their degree in polynomials may increase by

```
> ( e, q ) := PartialCylindricalAlgebraicDecompose( expr );
```

$$e, q := [[true, y = -3, x = RootOf\left(53\_Z^3 - 567\_Z^2 + 70\_Z - 2007, \frac{80397298677970683 2879}{73786976294838206464} \ldots \frac{643178389423765466 3059}{590295810358705651712}\right)]], true$$

```
> alias( R = op( [ 1, 3, 2 ], e ) ): # Let R be the long RootOf
> eval( GetUnquantifiedFormula( expr ), e[][ 2.. -1 ] );
```

$$53 * R^3 - 567 * R^2 + 70 * R - 2007 = 0$$

```
> evala( Normal( % ) );
```

$$0 = 0$$

Figure 3-10: Usage of `evala( Normal( ... ) )` to let Maple deduce the value of polynomials when witnesses with real algebraic numbers are substituted for variables. `evala( Normal( ... ) )` computes the normal form for $53 * R^3 - 567 * R^2 + 70 * R - 2007$ using the algebraic field extension for R.

the action of virtual substitution. In CAD, one must project and lift with respect to free variables, and in the example above this is true of the $n-1$ free variables.

A slight modification of the above case study highlights a different contrast. For the formula $\exists x_1\ g(x_1,\ldots,x_n)x_1 > 0 \wedge g(x_1,\ldots,x_n)x_1 < 0 \wedge f_1(x_1,\ldots,x_n) \wedge \cdots \wedge f_k(x_1,\ldots,x_n)$, VTS will *not* deduce *false* without a strong Tarski formula simplifier, while engaging with the unquantified variables $x_2 \ldots x_n$ will allow CAD to do so. This is a blessing and a curse for VTS, which essentially ignores free variables.

VTS benefits from strong simplification to produce simple quantifier free output, but CAD is in some sense its own simplifier, where the quantifier free output from CAD is a disjunction of conjunctions owing to descriptions of CAD cells. In particular, CAD will never fail to deduce when an input formula is equivalent to *true* or *false*, due to cylindricity and propagation of truth values, but VTS may produce formulae equivalent to *true* or *false*, but unfortunately unsimplified, such as $x > 0 \wedge x = 0$, which is actually equivalent to *false*. Other more nuanced potential output formulae include polynomials without real roots, such as $c^2 + c + 1 = 0$, which VTS cannot deduce as *false*. This is especially egregious for `QuantifierElimination`, which features no strong simplification for VTS. This brings `QuantifierElimination`'s QE output by CAD closer to "candid". On the other hand, quantifier free output produced from solely VTS is always a Tarski formula, due to VTS acting entirely on Tarski formulae, whereas CAD will produce an Extended Tarski formula including real algebraic functions, which are arguably sometimes more, and sometimes less intelligible than mere Tarski formulae.

VTS can be viewed as "recursive" in the sense that it can be used to eliminate one quantifier at a time, and the canonicalization of results of virtual substitution as `IQER`s in Chapter 2 reflects this — every node is implicitly an individual QE problem quantified with a number of quantifiers inversely commensurate with its level in the tree. CAD cells, which associate a tree structure, have no such equivalence to a QE problem. In QE, Tarski formula held by a CAD cell (tarski_formula) is merely the equivalent of the unquantified part of the top level input formula Φ at the full sample point of a cell, in order to attempt to deduce the truth of the formula at that cell. Once CAD commences on a QE problem, CAD must commence with projection strictly followed by lifting, due to the opposition in direction between the two (Figure 3-7), which makes CAD more "clunky" than VTS in some sense. This includes the evolutionary methods, each of which for CAD must proceed with some type of projection, before "correcting" the CAD tree, then lifting (Section 5.2), while for VTS one need only correct the tree before proceeding with more VTS.

On the other hand, the "tree corrections" of this work fall to a comparable tree traversal for both VTS and CAD alike. For example, there are many similarities between Algorithms 46 and 53. Both algorithms act upon the formula held by a `CADCell` or `IQER`, and as objects both are similar in the sense they are nodes of trees in differing contexts. This highlights that the objects can be seen as somewhat similar. Both inherit parenting to enable a tree structure and associate some property equivalent to the edge above it. In both cases the edge represents a point to substitute, although

for an `IQER` it is principally "virtual" (its testpoint), while for a `CADCell` it is a real algebraic number that represents that cell's local sample point (sample_point).

3.12 Algorithms

The top level algorithms for CAD, two of which are specifically for the framework of QE including VTS, are listed here. QEPCADL is named as such purely to differentiate from PartialCylindricalAlgebraicDecompose, and acts upon the QE problem defined by one `IQER`. Meanwhile, VTSToCADWhole acts upon the QE problem defined by several genuine leaf & ineligible `IQER`s at the termination of VTS with ineligible `IQER`s. These will be found useful in the following section, discussing the poly-algorithm.

Algorithm 37 QE by Partial Cylindrical Algebraic Decomposition

Input: $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi(x_1,\dots,x_n)$, a prenex quantified Real Tarski formula, and a myriad of options for CAD, including lcs, a set of lifting constraints, and open, a boolean flag dictating if Open CAD is admissible by the user

Output: QE output dependent on number of output arguments requested, up to and including the quantifier free equivalent of $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi(x_1,\dots,x_n)$, witnesses for the problem, and the `CADData` for the CAD

1: **procedure** PartialCylindricalAlgebraicDecompose($Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi$)
2: (A, E) ← getPolySets(Φ)
3: localopen ← OpenCAD **and** hasAllStrongRelations(Φ) ▷ Algorithm 19
4: quants ← $[Q_{n-m+1},\dots,Q_n]$
5: bases ← a projection data structure generated from A and E, with variable strategy potentially intrinsically tied to projection, hence generates a variable order vars ← $[x_1,\dots,x_n]$
6: cad ← an empty container
7: leaves ← an empty container
8: rootCell ← `CADCell`(Φ)
9: (constraints, bounds) ← parseLiftingConstraints(lcs)
10: Code Fragment 28, regulation CAD lifting by Partial CAD
11: Code Fragment 35, recovery from lifting failures
12: **return** QE output dependent on number of output arguments requested
13: **end procedure**

Algorithm 38 Full Cylindrical Algebraic Decomposition

Input: A formula Φ, **OR** sets of polynomials A and E, and a myriad of options for CAD, including lcs, a set of lifting constraints, and open, a boolean flag for requesting an Open CAD
Output: `CADData` for the CAD

1: **procedure** CylindricalAlgebraicDecompose(Φ)
2: (A, E) ← sets of polynomials from inequalities and equational constraints from input respectively (using Algorithm 21 if appropriate)
3: (constraints, bounds) ← parseLiftingConstraints(lcs)
4: bases ← a projection data structure generated from A and E, with variable sorting potentially intrinsically tied to projection, hence generates a variable ordering vars ← $[x_1, \ldots, x_n]$ (using CADChooseVarsProjection)
5: rootCell ← `CADCell`(*true*) ▷ Tarski formulae and truth values of cells irrelevant
6: cad ← an empty container
7: leaves ← an empty container
8: Code Fragment 29 ▷ Regulation full CAD lifting
9: **if** |curtains| > 0 **then** ▷ Need to attempt curtain recovery
10: decomposeCurtainCellsCAD(bases, rootCell, vars, n, leaves, curtains, open, constraints, bounds) ▷ Algorithm 30
11: **end if**
12: **return** `CADData` associated to all the CAD data structures, including the projection, and leaves of the CAD tree
13: **end procedure**

Algorithm 39 QE by Partial CAD on the QE problem defined by one `IQER`

Input: An `IQER` I, and a myriad of options for CAD
Output: `CADData` for the CAD

1: **procedure** QEPCADL(I)
2: (Ψ, C) ← PartialCylindricalAlgebraicDecompose($Qx_{n-m+1} \ldots Qx_{n-I\mapsto\text{level}}\ \Phi'(x_1, \ldots, x_{n-I\mapsto\text{level}})$ where Φ' is the unquantified formula associated to I ($I \mapsto$ formulaSimplified), Ψ is the quantifier free equivalent of I, and C is associated `CADData`
3: $I \mapsto$ cad_formula ← Ψ
4: **return** C
5: **end procedure**

Algorithm 40 QE by Partial CAD on the "whole" QE problem defined by termination of VTS, (4.1), without poly-algorithmic QE

Input: $Qx_{n-m+1} \dots Qx_{n-t}\ B\left(B_{j=1}^{k} I_j^{'}\ B_{j=1}^{s} I_j\right)$, the state of termination of VTS with ineligible `IQER`s via (4.1), and a myriad of options for CAD

Output: Ψ, quantifier free equivalent of (4.1)

1: **procedure** VTSToCADWhole($Qx_{n-m+1} \dots Qx_{n-t}\ B\left(B_{j=1}^{k} I_j^{'}\ B_{j=1}^{s} I_j\right)$)

2: $\Psi \leftarrow$ PartialCylindricalAlgebraicDecompose$\left(Qx_{n-m+1} \dots Qx_{t}\ B\left(B_{j=1}^{k} I_j^{'}\ B_{j=1}^{s} I_j\right)\right)$ where Ψ is the quantifier free equivalent of (4.1). $\triangleright$ t is the minimum level amongst all the `IQER`s

3: **return** Ψ

4: **end procedure**

Chapter 4

The Poly-algorithmic QE System

This section discusses the main bespoke feature researched for this thesis. The initial exploration of the ideas and methodology was in [64]. The reader may wish to watch an overview of the poly-algorithmic method featured in the publicly available video [66] first recorded for ICMS 2020.

4.1 From VTS to CAD

Section 2.3 discusses facts about distributivity of quantifiers into the implicit disjunction/conjunction formed by VTS in elimination of existential and universal quantifiers respectively. In particular, this forms a tree structure and a choice of `IQER`s within any one block of quantifiers. The intention of the poly-algorithm is to use VTS as far as possible (up to a specified degree ≤ 2), and we discuss the situation where VTS cannot complete quantifier elimination alone, due to receiving at least one ineligible `IQER`.

Because we can only distribute one type of quantifier through the disjunction or conjunction formed within one block, the poly-algorithm is only used within the last block of quantifiers. For many examples, the problem is homogeneously quantified anyway, so we are always in the last block, and in the context of QE for `QF_NRA`, certainly homogeneously quantified. One could distribute the (remainder) of the innermost block through the canonical boolean operator for VTS onto the ineligible `IQER`s, but usage of CAD on these would create cylindrical formulae to quantify with the remainder of the blocks of quantifiers, which is largely nonsensical, not least because CAD will view the variables from those remaining blocks as free, constructing geometry around them — the complexity of CAD is highly dependent on the number of free variables in a formula. Usage of the poly-algorithm in this context would hence "double up" on usage of variables from later blocks of quantifiers. To make matters worse, the quantifier free output of CAD in `QuantifierElimination` is generally an Extended Tarski Formula, which is not appropriate as input to any QE function in `QuantifierElimination`, so we cannot even consider "re-quantification" of such with the later blocks of quantifiers with the current implementation.

Hence the state of VTS on termination of VTS must be

$$Qx_{n-m+1} \dots Qx_{n-t} \; B \left(B_{j=1}^{k} I_j^{'} \; B_{j=1}^{s} I_j \right) \tag{4.1}$$

for t the minimum level of any $I_1, \ldots, I_s$, the $s > 0$ ineligible IQERs, and $I_1^{'}, \ldots, I_k^{'}$, $k \geq 0$ leaf IQERs[1], B is the canonical boolean operator corresponding to the quantifier symbol $Q \in \{\exists, \forall\}$.

1. Select an IQER I from amongst those ineligible $(I_1, \ldots, I_s)$, according to some metric. This work investigates a metric based on depth of the selected IQER. If the root IQER was not ineligible, then we obtain a choice of IQERs to solve with fewer quantified variables each, albeit potentially at higher degree in such variables. The depth of each IQER is inversely proportional to its number of quantified variables. Perform the QE $Qx_{n-m+1} \ldots Qx_{n-I \mapsto \text{level}}\ I$ via Partial CAD. We retain data for this CAD, C.

2. If this yields a meaningful truth value we are done. Else, select the next IQER I to solve via the chosen metric.

   - We expect that the IQERs of this tree have similar boolean structure and polynomials (Section 4.1).
   - By examining the polynomials from I as a flat set, we can measure the proportion of polynomials in this IQER with the polynomials from the formula of the last IQER used in the CAD. The formula from the last IQER exists at the root cell for the CAD via repurposing. We need not be as precise as to make distinctions about equational constraints here, due to the assumptions about similar boolean structure between IQERs (Section 4.1).
   - If the proportion meets some threshold, we reuse the CAD C incrementally to solve I (Section 4.1). If not, we can discard C and create a new CAD to solve I. Additionally, I must contain no free variables new to C for C to accommodate I, else we must create a new CAD. However, new quantified variables are allowable (Section 4.1).

3. And hence we iterate this process further, and each ineligible IQER uses the cad_formula property to store the equivalent ETF equivalent to itself for the purposes of full QE output later. Again, if we receive a meaningful truth value, we are done.

4. Upon termination of the poly-algorithm, QE output for this block of quantifiers can be described by $B\left(B_{j=1}^{k} I_j^{'}\ B_{j=1}^{s} I_j\right)$ where each I_j uses its cad_formula as its quantifier free equivalent, and each $I_j^{'}$ uses its formulaSimplified. This also describes QE output when $s = 0$.

Incremental CAD is comprehensively covered in Section 5.2 — incremental and not decremental technology is of relevance. CAD is always of two parts, projection and lifting, and incremental CAD reflects this similarly. The full incrementality of accommodating a new IQER from an existing CAD is Algorithm 44. This uses Algorithm

[1] $I_1^{'}, \ldots, I_k^{'}$ are only meaningful leaves if the user defines unusual termination criteria, i.e. setting the keyword option `'eagerness'` within `QuantifierElimination` to a low value in order to force full QE in general without early termination.

50 to handle incremental projection, followed by incremental lifting instigated by the bespoke Algorithm 52 followed by regulation Partial CAD lifting of Code Fragment 28. Importantly, the discussion for Algorithm 52 clarifies that it attempts to *repurpose* the CAD tree for the formula from the incoming `IQER`. This means re-evaluating the tarski_formulae of the cells from the tree to possibly rededuce their truth_values, but importantly much of the geometry can be reused due to the resulting projection sets being similar due to the "poly-share" threshold (Section 4.1).

Because usage of VTS assumes that the quantified input formula is a Tarski formula, i.e. of integral polynomial constraints, the poly-algorithm is only applicable (in a non-trivial sense) on Tarski formulae. One could canonicalize the root `IQER` as ineligible in the case of irrational numbers, but usage of the poly-algorithm is trivial whenever the root `IQER` is ineligible — the QE reduces to essentially exactly the same methodology as to if PartialCylindricalAlgebraicDecompose were called.

The best case for usage of the poly-algorithm is usage of CAD on exactly one ineligible `IQER` to receive a meaningful truth value. In the case of fully (homogeneously) quantified formulae, *true* and *false* are the only candidates for quantifier free equivalents of `IQER`s. In this way, there is a view for the poly-algorithm to cater well to the case for fully quantified formulae. When QE is used beneath SMT solvers for `QF_NRA`, the formulae are informally viewed to be fully existentially quantified, and hence there is a hypothesis that the poly-algorithm accommodates `QF_NRA` well via finding *true* (i.e. satisfiability) without using all `IQER`s.

Incremental CAD Variables with Traversal of the VTS tree

Depth-wise traversal of the VTS tree in the context of the poly-algorithm means that the depths of the selected `IQER`s to repurpose the held CAD are decreasing (perhaps not monotonically). An `IQER` of strictly lesser depth than the last holds strictly more variables that are quantified. Referring to Figure 3-7, we understand that:

- we project with respect to the last variable in the ordering for a CAD first,
- and the direction of lifting is opposite to that of projection, hence the children of the root cell are with respect to the first variable in the ordering for the CAD.

In fact, it is convenient that VTS acts in the opposite direction to CAD's projection, meaning that the direction of construction of the trees of each algorithm actually coincide.

Say that the first `IQER` to be used in CAD is of level $0 < j < m$ in a fully homogeneously quantified problem $Qx_{n-m+1} \dots Qx_n \ \Phi(x_1, \dots, x_n)$. The CAD hence is of the variables $x_1, \dots, x_{n-j}$. To accommodate an `IQER` of level $j < k < m$, we must now extend the CAD to include the variables $x_{n-j+1}, \dots, x_k$. But:

- projection on polynomials including these variables is canonical, being much as it would be in the standard case, and the produced polynomials in $x_1, \dots, x_j$ can be fed through the rest of projection incrementally (by caching — Section 5.2.1),

- the new variables being such that they can appear at the end of the ordering means that the lifted CAD tree need only be *extended*, and not "*relifted*". The existing geometry is all valid, with the admission that the cells need be repurposed in terms of the incoming Tarski formula of the `IQER` to evaluate.

The added variables $x_{n-j+1}, \ldots, x_k$ in the above context are new to that CAD. It already has an ordering for $x_1, \ldots, x_{n-j}$ induced by the name of the variable strategy supplied to QuantifierEliminate. Usage of the "projection" based orderings such as the "greedy" ordering, ndrr, or sotd are largely nonsensical to sort $x_{n-j+1}, \ldots, x_k$, requiring full projection of all orders in some context (Section 3.8), and both the projection and ordering created by this are unlikely to be intelligible in terms of the existing projection, which we desire to extend incrementally. However, any heuristic inducing a comparison function on variables *is* amenable to be used to sort the incoming $x_{n-j+1}, \ldots, x_k$ in terms of the polynomials $\subset \mathbb{Z}[x_1, \ldots, x_k]$. Therefore the poly-algorithm obliges the supplied variable strategy if it is the Brown heuristic or ECHeuristic, else sorts these variables via the ECHeuristic on the polynomials for the new `IQER`.

Therefore, extension of a CAD to one in further *quantified* variables is entirely canonical. Inclusion of new *unquantified* variables would work at the opposite ends of projection and lifting respectively, and so importantly, would require a "re-lift" of the CAD, because prepending variables means that the children of the root cell are no longer intelligible in terms of this ordering. Deduction and collection of new unquantified variables in an `IQER` is done at the same time as evaluation of the "poly-share criteria" for convenience in deciding whether to create a new CAD or not, as additional free variables precludes the existing CAD to be used. Extension of a CAD is an exclusive concept to depth-wise traversal, because usage of breadth-wise traversal includes all possible variables to the CAD as early as possible (and similarly further free variables are not amenable to inclusion to the CAD via incrementality).

Usage of both depth-wise and breadth-wise traversal of the VTS tree amongst ineligible `IQER`s imposes some restriction on variable ordering. Breadth-wise traversal fixes the ordering of all possible variables to be included in the CAD as soon as possible, at the cost of a CAD as "large" as possible. Depth-wise traversal aims to create as small a CAD as possible with the aims of potentially finding a meaningful truth value for the overarching quantifier Q, with some scope but not complete freedom for ordering future variables.

In contrast to the discussion on variable orderings for the poly-algorithmic approaches here, collapsing of the whole VTS tree to one quantified formula for CAD to solve imposes very little restriction on variable ordering. In particular, if I_1 is the deepest `IQER` of level i, and I_2 is the lowest `IQER` of level $0 < j < i$, then in using this methodology CAD has no restriction on the ordering of $x_{n-i}, \ldots, x_{n-j}$ — it need not correspond to the ordering of them as used within VTS, as CAD is purely a means to an ends as of this point, and none of the CAD data is directly used in terms of correspondence with VTS in terms of e.g. meaningful witnesses.

The "Poly-share" Criteria

The "poly-share criteria" refers to the criteria used to decide whether to reuse a CAD to solve an incoming `IQER`.

Technically, the most reliable way to examine whether the geometry for the CAD accommodates the incoming `IQER` well is to examine the projection bases, however this has an intrinsically exponential number of polynomials spread throughout all levels of the structure. Due to the assumption that the polynomials are similar between `IQER`s, we instead suggest to take GCDs between the polynomials of the incoming `IQER` and those from the *last `IQER` used*. As this is the point where we first examine the polynomials of the `IQER`, we also ensure that no free variables are contained within the `IQER` to preclude us from reusing the existing CAD via the discussion in the last paragraph. The formula from the last `IQER` is stored at the root cell of the CAD via the repurposing of Algorithm 52. In decomposing the formula of the last and incoming `IQER`s to sets, we can take GCDs and deduce the proportion of polynomials from the incoming `IQER` that have a nontrivial GCD with the current formula for the CAD. The decompositions of both are to one flat set, even if one could decompose each to two including a set of equational constraints (such as output from Algorithm 21), but such distinctions are unlikely to be very meaningful. The function decomposeTFAsSet decomposes a (real) Tarski formula to one flat set of all the polynomials contained within, unlike getPolySets which identifies and returns a set of ECs. checkPolyIntersection makes no distinction as to equational constraints from the CAD or the `IQER`, because of the assumption that boolean structure between `IQER`s is likely to be similar. Inspection of the Tarski formula held by the root cell as opposed to the projection sets also avoids the difficulty induced by examining ECs in projection that have been modified from the formula due to usage of Gröbner bases.

The return value of checkPolyIntersection is checked against a threshold value such that I is believed to be well accommodated by C. This threshold value is defined by `POLY_SHARE_THRESHOLD`. This value between 0 and 1 is a macro defined by `QuantifierElimination`, and as such is a variable set at "compile time" for the package. Extremal values of `POLY_SHARE_THRESHOLD`, i.e. those very close to 0 are 1 are ostensibly obviously poor for efficiency, with the former having us repurpose CADs sharing extremely little geometry with that required to solve the incoming `IQER`, and the latter having us completely recompute the held CAD for an `IQER` that is very likely to be well accommodated by the CAD. The value of `POLY_SHARE_THRESHOLD` in `QuantifierElimination` is $\frac{1}{2}$. The case studies on the poly-algorithm in Section 7.3 seem to suggest that when a CAD is reused within the poly-algorithm, it is always reused, with very few new polynomials needing to be introduced in any projection bases, let alone at the top level bases. An investigation as to the optimal value `POLY_SHARE_THRESHOLD` is interesting further work. The aforementioned case studies imply that values of `POLY_SHARE_THRESHOLD` below 0.5 are likely to lead to exactly the same behaviour, suggesting investigation of higher values. One notes that recreating the CAD from scratch removing redundancy of geometry is more likely to avoid lifting failures due to convoluted (i.e. involving highly nested RootOfs) redundant geometry, at the cost of rebuilding new geometry that may have already existed in the held CAD

from scratch. The case studies of Section 7.3 always have that the CAD generated in poly-algorithmic QE is constantly reused, which implies that a `POLY_SHARE_THRESHOLD` of $< \frac{1}{2}$ would change nothing with respect to the examples currently found where the poly-algorithm is relevant, but a larger value may result in more finer control of re-usage of the CAD, either to performance's benefit or detriment.

Open Problem 70. *Is there a more optimal value than $\frac{1}{2}$ for `POLY_SHARE_THRESHOLD`, or even a better replacement for the "poly-share criteria"?*

Algorithm 41 Evaluation of the "Poly-share" Criteria

Input: C previously used `CADData`, I an ineligible `IQER`, and VTSVars an `Array` of all quantified variables from a last block of quantifiers
Output: A value between 0 and 1, the ratio of similarity of the polynomials from the `IQER` to those from C

```
1: procedure checkPolyIntersection( C, I, VTSVars )
2:     F ← decomposeTFAsSet( I ↦ formulaSimplified )
3:     G ← decomposeTFAsSet( C ↦ RootCell ↦ tarski_formula ) where
           C ↦ RootCell is the root cell for the CAD C
4:     if F contains variables not contained in G or VTSVars then
5:         return 0      ▷ F contains new unquantified variables — all variables from
               VTSVars are guaranteed to be quantified
6:     end if
7:     c ← 0
8:     for f in F do
9:         for g in G do
10:            if deg(gcd(f, g)) > 0 then
11:                c++
12:                break
13:            end if
14:        end for
15:    end for
16:    return c/|F|
17: end procedure
```

Comparison to Incrementality via Clauses

Despite the particular approach, the work acknowledges that the formula (4.1) is amenable to usage of "standard" incrementality acting upon the inner clauses due to the induced distributivity, and VTSToCADWhole (Section 4.3) does not attempt this approach — it is not incremental in any sense beyond delayed substitution of sample points into formulae, which is relevant considering the structure of (4.1), but the more relevant notion of incrementality for CAD is on projection polynomials (Chapter 5). Most generally, (4.1) should be a disjunction of conjunctions for an existentially quantified problem, or a conjunction of disjunctions for a universally quantified problem.

This approach intends to differ to any other less bespoke incremental CAD approach on clauses in a few ways:

- this framework associates the existing CAD to exactly one `IQER` at a time, enabling a one to one correspondence between a meaningful leaf `CADCell` and the root `IQER` for the last block of quantifiers and hence the production of meaningful witnesses from usage of both VTS and CAD to achieve QE (Section 4.4.1),
- the levels of `IQER`s define strategy for selection of "clauses", where we know that extension of a CAD to accommodate an `IQER` of lesser depth is entirely canonical due to previous paragraphs, and we only need check for the existence of new unquantified variables, as they do not largely permit CAD incrementality in terms of the CAD tree, which would need relifting in this case (and as such we may as well regenerate a fresh CAD). We aim to solve the deepest `IQER`s first in order to find a meaningful truth value as early as possible.
- by assuming that nearby `IQER`s are similar in terms of their polynomials and boolean structure, we acquire scope to compare "similarity" between `IQER`s (the "poly-share criteria"), which becomes part of the strategy for discarding or retaining the held CAD to solve the next `IQER`. The specific methodology for CAD incrementality to "repurpose" for the `IQER` is also unique and delineated in detail in Chapter 5.

While `QuantifierElimination` uses the developed Lazard projection & lifting CAD as the incremental CAD to use beneath the poly-algorithm, there is the possibility of usage of any incremental CAD being available as an option. We note `RegularChains`' CAD has incremental technology. However, the incrementality is at the least currently not exported to the extent it is controllable by a user in the sense of evolutionary methods (Definition 71). In fact, the poly-algorithm currently assumes "repurposing" of CADs, parallel to incrementality. Retention and control of a CAD data structure is the minimum required to imitate the poly-algorithmic methodology with respect to CAD. Beyond that, it is unknown to the author to what extent the bespoke `RegularChains` methodology could accommodate new quantified or unquantified variables appearing from successive `IQER`s to solve (Section 4.1). The discussion of Section 4.1 does however remain true for any projection & lifting CAD.

Code Fragment 42 actually features similar "lifting failure avoidance" as that of CAD (Section 3.7.2). This lifting failure avoidance can be viewed as more of a "global" lifting failure avoidance, where the CAD lifting failure avoidance is "local". The avoidance means that we can ignore failure to evaluate one `IQER` via CAD if we receive a meaningful truth value from another, similar to the way CAD attempts to ignore lifting failures by propagation of truth values from other cells with meaningful truth values. Propagation of truth values in VTS is only ever trivial, considering we are always homogeneously within one block, so "propagation of a meaningful truth value" would only ever propagate that meaningful truth value directly to the root `IQER` immediately. In any case, failure to evaluate a CAD on an `IQER` still comes at the (time) expense of trying to do so, however we can still try to maximise success of QE as is the usual aim.

Fragment 42 Poly-algorithmic QE on ineligible `IQER`s

```
1: if i = −1 then                                   ▷ iqers is only of ineligible IQERs
2:     if 'HybridMode' = 'whole' then
3:         return VTSToCADWhole( Ψ ) where Ψ is the QE problem defined by the
               ineligible IQERs remaining in iqers, the leaf IQERs in leaves, and the
               canonical boolean operator for Q, B — i.e. (4.1)
4:     end if
5:     i ← VTSToCADStrategy( iqers, 'HybridMode' )
6:     problemCADs ← an empty container
7:     repeat
8:         iqer ← pop( iqers, i )
9:         try
10:            if C is empty CADData or checkPolyIntersection( C, iqer, vars ) <
                   POLY_SHARE_THRESHOLD then
11:                Clear the caches of all polynomial operations (i.e. discriminants and
                       resultants)          ▷ As per discussion on incremental projection,
                       Section 5.2.1
12:                C ← QEPCADL( iqer )
13:            else
14:                modifyCADResult( C, iqer, Q, vars )
15:            end if          ▷ In either case where the CAD succeeds, iqer receives a
                       cad_formula
16:        catch CAD_EXCEPTION_STRINGS:
17:            Add [iqer, E] to problemCADs, where E is the exception caught
18:        end try
19:        if iqer ↦ cad_formula is a meaningful truth value for Q then
20:            Remove all IQERs from iqers and all lists from problemCADs
21:        end if
22:    until ( i ← VTSToCADStrategy( iqers, 'HybridMode' ) ) = 0
23:    if |problemCADs| > 0 then
24:        Produce an ERROR from any exception stored in problemCADs,
               preferably any one about curtains
25:    end if
26: end if
27: return QE output dependent with number of outputs requested
```

Programmatically, problemCADs is the "global" equivalent of the problemCells container used in "local" Partial CAD. `IQER`s that fail evaluation by CAD are stored with their corresponding exception string, and we can reraise this error later if we fail to find a meaningful truth value elsewhere. Nothing bespoke is required to deduce which `IQER`s to remove from the tree in the same sense as Algorithm 15, because of the homogeneity of quantifiers & their corresponding meaningful truth values as above.

Algorithm 43 Poly-algorithmic Quantifier Elimination via VTS into CAD on a homogeneously quantified formula

Input: $Qx_{n-m+1}, \dots, Qx_n\ \Phi(x_1, \dots, x_n)$, a prenex quantified Tarski formula
Output: QE output depending on output arguments requested — up to and including the quantifier free equivalent Ψ to $Qx_{n-m+1}, \dots, Qx_n\ \Phi(x_1, \dots, x_n)$, the witnesses for equivalence of Ψ to $Q_{n-m+1}x_{n-m+1} \dots Q_n x_n\ \Phi(x_1, \dots, x_n)$, and a data structure `QEData` for the quantifier elimination allowing for further evolutionary operations including the CAD via `CADData`, if a CAD was produced (and hence retained) during Code Fragment 42

1: **procedure** QuantifierEliminate($Qx_{n-m+1}, \dots, Qx_n\ \Phi(x_1, \dots, x_n)$)
2: rootIQER ← `IQER`(Φ)
3: iqers ← a container containing rootIQER
4: leaves ← an empty container
5: VTSVariableStrategy(rootIQER, vars) ▷ First pass of variable strategy in terms of the root `IQER` to maximise chance `IQER` is viewed as non ineligible (Section 2.3.2)
6: Code Fragment 4 to propagate VTS on non ineligible `IQER`s
7: Code Fragment 42 on the ineligible `IQER`s iqers, if any exist
8: **return** QE output corresponding to the number of output arguments requested
9: **end procedure**

Proof of usage of modifyCADResult to successfully repurpose the CAD C to solve I is deferred to Proposition 81 in Section 5.2, such that the intermediate algorithms necessary for CAD incrementality are explained first.

Boolean Structure of `IQER`s, and Equational Constraints

The original rationale for investigation of a poly-algorithm in this way is the preservation of boolean structure of the formula for QE under virtual substitutions on a formula. In the context of this work, the hypothesis is that the `IQER`s of a VTS tree "look" similar, especially those closer to each other in the tree. This is both with respect to the polynomials within their relations, and their boolean structure in terms of the relational operators and overall boolean skeleton structure (in terms of `And` and `Or`). One notes that:

- Usage of pure linear VTS without infinitesimals always preserves relational operators. Substitution of a test point T with polynomial p, $\deg_x(p)$ into a relation

Algorithm 44 Modification of a CAD to accommodate another `IQER`

Input: C `CADData` for a CAD to repurpose, I an `IQER` to solve using modification of C, Q the quantifier for the overarching block of quantifiers, vars an `Array` of all *quantified* variables $[x_{n-m+1}, \ldots, x_n]$ for the system, a myriad of CAD keyword options such as "UseGroebner", "UseEquations", "PropagateECs" related to projection, and OpenCAD, a boolean flag via keyword option representing if the user requests only sectors to be lifted

Output: No meaningful return, but I gets a quantifier free equivalent via assignment of its cad_formula, and C modified inplace

1: **procedure** MODIFYCADRESULT(C, iqer, Q, vars, UseGroebner, UseEquations, PropagateECs, OpenCAD)
2: Let $\Psi =$ iqer $\mapsto$ formulaSimplified
3: (A, E) $\leftarrow$ getPolySets(Ψ)
4: Let $C \mapsto$ Vars be the variables associated to C, with $C \mapsto$ numVars and $C \mapsto$ numQuants the total number of variables and quantified variables respectively for C
5: $d \leftarrow \max($ $n - I \mapsto$ level $- C \mapsto$ numQuants, 0) ▷ Difference in number of quantifiers
6: $C \mapsto$ numVars $+= d$
7: $C \mapsto$ numQuants $+= d$
8: Sort vars$[-I \mapsto$ level $- d \ldots - I \mapsto$ level $- 1]$ $([x_{n-I\mapsto\text{level}-d+1}, \ldots, x_{n-I\mapsto\text{level}}])$ by an appropriate heuristic such as the Brown heuristic or ECHeuristic using A and E, and extend $C \mapsto$ Vars with the result
9: Let $C \mapsto$ RootCell be the root `CADCell` associated to the `CADData`, and $C \mapsto$ Bases be the `projection` object for the CAD
10: (incBases, newPivots) $\leftarrow$ projectionIncremental($C \mapsto$ Bases, A, E, $C \mapsto$ Vars, $C \mapsto$ numVars, d, 0, UseGroebner, PropagateECs, UseEquations)
11: cad $\leftarrow$ an empty container
12: leaves $\leftarrow$ an empty container
13: problemCells $\leftarrow$ as an empty container
14: localopen $\leftarrow$ OpenCAD **and** hasAllStrongRelations(Ψ)
15: **if not** localopen **and** Open CAD used before for data **then**
16: **ERROR** — the lifting may fail to deduce QE due to lack of sections (proof of Proposition 83)
17: **end if**
18: traverseCADTreeModify($C \mapsto$ RootCell, cad, leaves, problemCells, $[Q, \ldots, Q]$, $C \mapsto$ Vars, $C \mapsto$ numQuants, $C \mapsto$ numVars, incBases, Ψ, localopen, newPivots, $C \mapsto$ Bases)
19: Code Fragment 28 (regulation Partial CAD lifting), applied in an evolutionary context to relevant properties of data
20: Code Fragment 35 (lifting failure recovery by curtain decomposition), applied in an evolutionary context to relevant properties of data

Algorithm 44 Modification of a CAD to accommodate another `IQER`, Part 2

21: **for** leaf **in** data $\mapsto$ Leaves **do**
22: leaf $\mapsto$ iqer_pointer $\leftarrow I$ $\triangleright$ For potential witness production later (via Code Fragment 45)
23: **end for**
24: $I \mapsto$ cad_formula $\leftarrow$ the quantifier free equivalent of I, deduced from the leaves of the CAD tree (leaves)
25: **return**
26: **end procedure**

$f\,\rho\,0$ reduces to returning $\text{prem}_x(p, f)\,\rho\,0$, hence the relational operator is retained [43, Algorithms vs-at, vs-prd-at, pseudo-sgn-rem]. The same is not true when infinitesimals are present in the test point, which produce different formulae with particular structure [43, Algorithms expand-eps-at, vs-inf-at]. Via the recursion of virtual substitution in Algorithm vs-at to act merely upon atoms, we also preserve boolean structure of the formula before substitution (sans collapsing of formulae due to simplification).

- Usage of structural test points of non-linear degree without infinitesimals can also preserve relational operators. If T is a test point with polynomial p of degree at least 2 in the variable x, then to substitute T into $f\,\rho\,0$, we compute $h := \text{prem}_x(p, f)$, which is of degree at most 1 (vs-prd-at). If $\deg_x(h) = 0$, we return $h\,\rho\,0$, but otherwise we must use the relevant formula scheme from [43, Appendix A] to describe substitution of T into $h\,\rho\,0$. One notes that amongst all the formula schemes for describing substitution of a quadratic root into a linear polynomial (Appendix A.2), the yielded formula features a relation with an operator corresponding to the operator of $f\,\rho\,0$, i.e. ρ. Importantly, this is always true when the relation was $f = 0$.

- For the hypothesis that the polynomials between `IQER`s can be similar, consider substitution of structural test points owing to the same polynomial, but with respect to different real types and/or real root numbers. Again looking at Appendix A.2. of [43], substitution of similar test points lead to very similar looking formulae via the schemes given. Often the formulae from the schemes are the same sans different relational operators leading to different sign conditions on the polynomials, such as $2aa^*b^* - a^{*2}b \geq 0 \wedge ab^{*2} + a^{*2}c - a^*bb^* = 0$ vs. $2aa^*b^* - a^{*2}b \leq 0 \wedge ab^{*2} + a^{*2}c - a^*bb^* = 0$ that are formula schemes for virtual substitution of the two different real roots of a quadratic of relevant real type into a linear polynomial (a, b, a^*, b^* are various coefficients, assumed to be in other parameters), which differ only in terms of the relational operator $\geq$ in the former replaced by $\leq$ in the latter. In these cases, with respect to just these formulae, CAD only needs to repurpose the lifted tree to reevaluate formulae, because the required polynomials to deduce the sign invariance already exist in projection. This means `IQER`s close together in the tree owing to structural test points with the same polynomial should have similar boolean structure and/or polynomials.

Importantly, the first two points donate interest in equational constraints on IQERs, which can be used in CAD on any one IQER in some context in the poly-algorithm. In particular, if the homogeneously existentially quantified formula for QE has equational constraints via being an existentially quantified conjunction, we may receive equational constraints in the ineligible IQERs to process via CAD in the poly-algorithm, and our interest in equational constraints in restricted projection in CAD comes to fruition. There is one further context where we may receive equational constraints at an IQER.

One notes the usage of "guards" in VTS. These formulae produced as a function of a structural test point are conjuncted with the result of virtual substitution of such a test point into a formula in the existential case. These guards are often sign conditions on coefficients of the polynomial from a test point, or the polynomial's discriminant, to assert the real type of such a polynomial. However, one notes that guards can yield equational constraints, i.e. equations manifesting at the top level of a conjunction, if one is to consider the a reductum of a polynomial. For example, to consider linear test points arising from a polynomial $ax^2 + bx + c$, one must assert that $a = 0$, hence this atom appears in the conjunction arising from the virtual substitution of such a linear test point. This gives more rationale to usage of CAD on such IQERs, due to the interest in equational constraints to restrict projection operations.

Reuse of Equational Constraints

Generally, an existentially quantified formula has the outer operator $\bigwedge$, and a universally quantified formula has the outer operator $\bigvee$. However, propagation of VTS creates formulae with a canonical boolean operator linked with the eliminated quantifier. In the purely existential case, VTS has formed a disjunction of conjunctions (or relations) where VTS preserves boolean structure of the input formula Φ. Guards must be conjuncted with the results of virtual substitution, so we know that the IQERs are still conjunctions (or relations after simplification). By allowing CAD to act upon individual IQERs, CAD is once again able to identify equational constraints of the form 1 from the examples of ECs in Section 3.7 in each IQER (often the same equational constraint(s) each time), as opposed to receiving a disjunction which is unlikely to yield equational constraints.

When discussing ECs, curtains are always a point of contention. One notes that in building and repurposing a CAD in the sense of the poly-algorithm, any one curtain need only be identified once by Code Fragment 27. Further, any curtain can only be associated with one pivot set, and we go to some length to ensure pivot sets are static and reused (Algorithm 50). New pivots can of course be introduced in solving new IQERs. If a curtain undergoes further decomposition after being identified as a point curtain or via Algorithm 34, its finer decomposition remains, and Algorithm 52 makes sure to only identify existing cells in the tree as curtains when a new pivot set of the appropriate level is introduced. Hence, cells can only ever be identified as a curtain at most once. In particular, any level $n-1$ curtain only need enter Algorithm 34 at most once per CAD, although regrettably it is conceivable that we traverse Algorithm 34 per each repurposing for newly produced level $n-1$ curtain cells each time. This would involve full projection of all orders each time, but such a case is likely very rare. Section

7.4.4 includes examples of curtain decomposition in the context of QE by Algorithm 32, in particular with respect to the poly-algorithm. Each time the decomposition occurs only in the first usage of CAD via QEPCADL.

The case studies on usage of the poly-algorithm (Section 7.3) seem to corroborate that the large benefit of repurposing of a CAD tree for various `IQER`s is reusage of equational constraints in the existential case. Of course, the presence of these ECs is likely as a result of the previous paragraph.

Satisfiability of Formulae & Poly-algorithmic Witnesses for QE

The intention behind selecting the most deep ineligible `IQER`s for CAD first is to receive a meaningful truth value on a CAD using the least variables possible, due to CAD's dependence on the number of variables. The number of variables in that CAD is dependent on the depth of the `IQER`, which reflects how many quantified variables remain in that `IQER`, although its depth says nothing about the number of free variables contained in it. In any case, minimizing the number of variables in the CAD before receiving a meaningful truth value to terminate QE is a win over any other approach which does not take the number of variables into account.

For fully existentially or universally quantified problems, the poly-algorithm associating a CAD to exactly one `IQER` at a time means that there is a one to one correspondence between a meaningful leaf cell and an ineligible `IQER`. This enables the production of meaningful witnesses after usage of the poly-algorithm, because CAD can provide the back substitutions to enable the "triangularity" of Theorem 23 for the ineligible `IQER` that CAD found a meaningful leaf cell below. Because we only ever retain one CAD at once in the poly-algorithm, and this CAD corresponds to at most one `IQER` at any one time, we know that upon termination of QE implied by production of a meaningful truth value in CAD means that this CAD corresponds to the correct `IQER` to process for witnesses. This is discussed and justified in Section 4.4.1.

"Global" Avoidance of Lifting Failures in CAD

As discussed following Code Fragment 42, the poly-algorithm attempts to avoid lifting failures from the CAD used for solution of successive ineligible `IQER`s. This means the CAD features its own local avoidance *and* recovery (where the recovery is in terms of curtains), but the poly-algorithm features its own global avoidance. This is because if we find a meaningful truth value from any `IQER`, we are sure to meet termination criteria, and are able to "ignore" failures from other `IQER`s. While most lifting failures are judged to be "temporary" (Section 3.7.2) in terms of the early development of `QuantifierElimination`, where further development of low level operations will make them less relevant, or even irrelevant, low level curtains in usage of multiple equational constraints are a more permanent type of lifting failure until further research provides "recovery" in this aspect within CAD, along the lines of [56]. Therefore, as per the "local" avoidance and recovery for CAD accommodates avoidance of at the very least low level curtains (and hopefully in the future, at most), the global avoidance follows the same philosophy. Various case studies of Section 7, and more broadly the QE

benchmarks in general seem to suggest that despite usage of multiple equational constraints beneath CAD within the poly-algorithm, low level curtains never cause non trivial usage of poly-algorithmic QE to fail by error, at least to an observable extent. We note the discussion of Section 4.1 with respect to curtain decomposition, and the associated case studies. Therefore `'UseEquations' = 'multiple'` has remained the default option used beneath poly-algorithmic QE, particularly in the benchmarking of this work. One notes that the incrementality of CAD used within the poly-algorithm pays close attention to whether cells are curtains after incremental operations (particularly evident in the proofs of Algorithm 52 with Algorithm 51).

If we fail to find a meaningful truth value with failed `IQER`s, we prefer to reraise an exception about a Lazard curtain, because it is a "mathematical" error. In such a case, one *can* restart the computation with `'UseEquations' = 'single'`, but again the case studies suggest this is likely rare, or at least such a case is not known.

The flexibility of the poly-algorithm could allow for restarting or correcting individual CADs on particular `IQER`s where low level curtains prevented the CAD from deducing a quantifier free equivalent, where the CAD would be restarted with `'UseEquations' = 'single'`, but such a feature is not implemented thus far. More likely, this feature would be deferred in favour of further research providing methodology to complete usage of multiple ECs for the Lazard projection in the manner of [56].

Section 7.3 delineates some case studies of usage of the poly-algorithm on various examples from the example sets of this project, in contrast to usage of "standard" CAD on the formulae formed by intermediate usage of VTS.

4.2 Strategy

VTSToCADStrategy controls strategy for selection of the next `IQER` to use CAD on. It uses the keyword option within top level QE, `'HybridMode'`, as a metric for this selection. In the same way as `'mode'` controls standard VTS `IQER` selection strategy on non ineligible `IQER`s, `'HybridMode'` takes the symbol `'breadth'` or `'depth'` to control breadth or depth-wise traversal of the VTS tree within the poly-algorithmic code fragment, but the value of `'HybridMode'` need not equal `'mode'`. This actually enables a greedy `IQER` selection strategy in terms of the poly-algorithmic code fragment, where one can select `IQER`s of the greatest depth in order to receive meaningful truth values as fast as possible for fully quantified formulae. In particular, it is greedy in the sense that it does not attempt to use checkPolyIntersection (Algorithm 41). One notes that that algorithm takes several (potentially multivariate) GCDs amongst polynomials already inherited from (potential) use of virtual substitution (both in the CAD and from the incoming `IQER`), and so calling it on every `IQER` in the container of (a potentially exponential number of) ineligible `IQER`s would be very costly as a strategy. Therefore `QuantifierElimination` offers greedy strategies either to look for meaningful truth values as fast as possible via 'depth'-wise traversal, or to accommodate as much of the geometry in as many variables as possible while still reusing equational constraints via 'breadth'-wise traversal. The benchmarking of Section 7.4.4 investigates usage of breadth-wise vs. depth-wise traversal of the VTS tree in the sense of the poly-algorithm.

The only scope for variable strategy is usage of heuristics between the Brown and

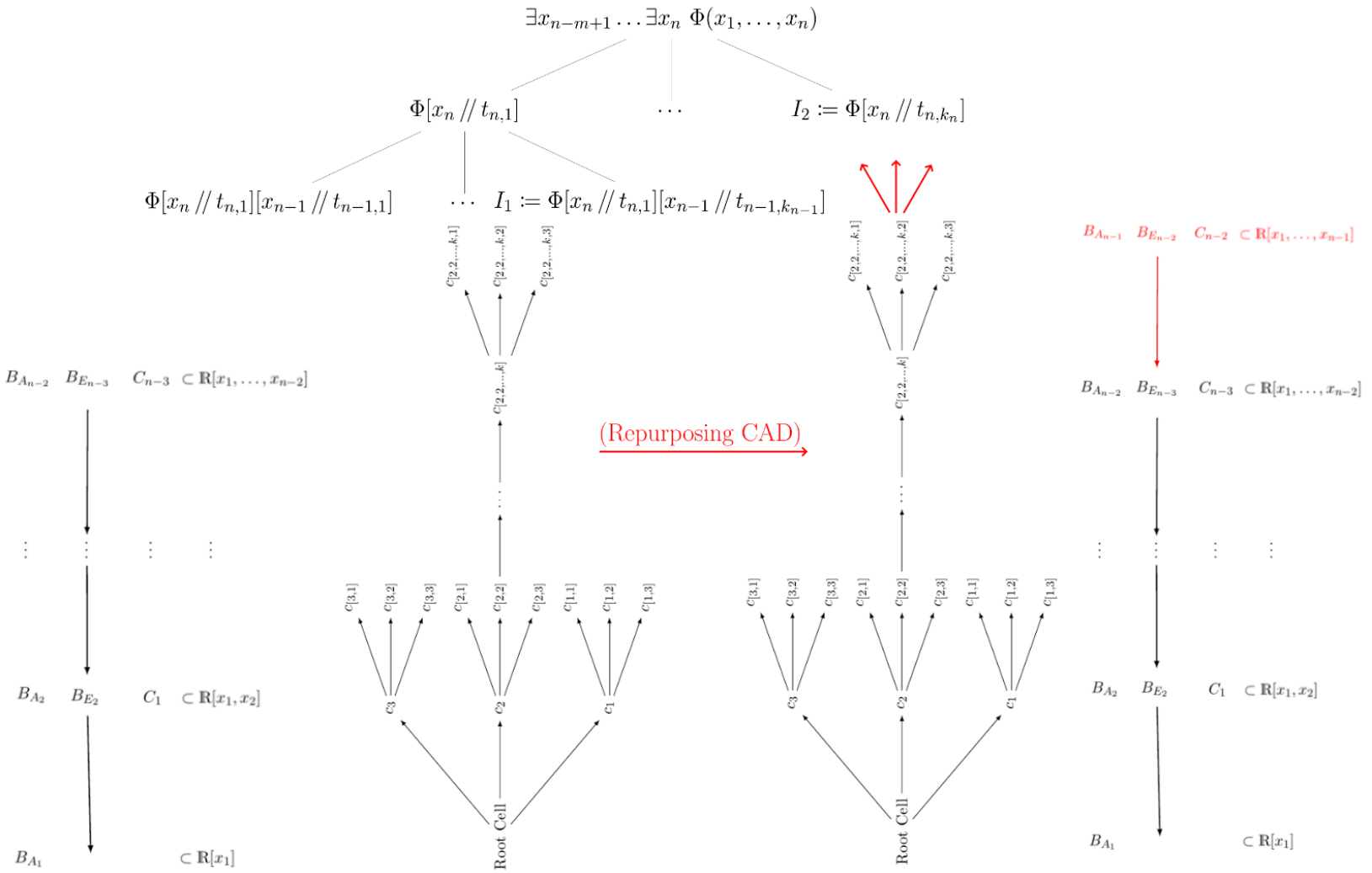


Figure 4-1: Figure demonstrating the trees involved in poly-algorithmic QE between VTS and CAD. Anything in red represents the extension of the CAD via the repurposing from the `IQER` I_1 to the `IQER` I_2, which are both asserted to be ineligible. In particular, the directions of VTS tree growth coincides with the direction of projection, which opposes the direction of CAD lifting. One prepends to the projection levels in CAD extension, and in principle appends to the CAD lifting tree after repurposing it via Algorithm 52. Everything in red is instigated by Algorithm 44. The `IQER` corresponding to $\Phi[x_n // t_{n,1}][x_{n-1} // t_{n-1,1}]$ being of the same level as I_1 would require no such "extension".

ECHeuristic when prepending new variables for the CAD when traversing the tree depth-wise (Section 3.8).

4.3 Standard Usage of QE by CAD

In contrast to usage of the poly-algorithm, the alternative is to pass the state of VTS on termination (4.1) as a formula to CAD in a "standard" way. This option is available by passing the symbol `'whole'` to the keyword option `'HybridMode'` for top level QE functions in `QuantifierElimination`. Therefore, doing so discards usage of the poly-algorithm. Programmatically Algorithm 40 is used in this case on the "whole" problem defined by the VTS tree. Usage of this methodology is benchmarked against the poly-algorithm in Section 7.4.4. One notes that one advantage of this non incremental approach is that CAD has all information available at the first point of asking — there are no restrictions in terms of variable ordering in initial CAD calls that may not cater well to later incremental calls. Furthermore, variable strategy has no requirement to have any relation to that from VTS at all, because usage of CAD in this case is purely a "means to an end" to provide a quantifier free equivalent. While there is no notion of typical incrementality in terms of projection polynomials, delayed substitution of sample points via inertisation (Section 3.4.2) helps the approach to lessen the cost of substitution of sample points into formulae in evaluation of cells, noting the structure of (4.1).

Usage of Other Leaves of the VTS Tree

From (4.1), we have that $I'_1, \ldots, I'_s$ are leaves of the VTS tree. Assuming that they are not meaningful leaves (due to abandonment of early termination criteria by specification of another keyword option), then $I'_1, \ldots, I'_s$ are `IQER`s equivalent to quantifier free formulae in free variables $x_1, \ldots, x_{n-m}$. As a result, the distribution of $Qx_{n-m+1}, \ldots, Qx_t$ through B is only non-trivial when acting on $B^s_{j=1} I_j$, which are implicitly quantified due to not being leaf `IQER`s. Due to the weak simplification deployed on formulae in VTS, while $I'_1, \ldots, I'_s$ are leaves, they could hold formulae that are in practice equivalent to *true* or *false* after strong simplification, but we would not know. In fact, they could then be meaningful leaves despite early termination criteria. The inclusion of $B^s_{j=1} I'_j$ within the formula to pass to CAD in a "standard" sense is then a matter of whether these operands are to be simplified or not, as CAD is in some sense its own simplifier when producing Extended Tarski formulae to describe QE output. `QuantifierElimination` chooses to include $B^s_{j=1} I'_j$ in the formula to pass to CAD when `'HybridMode'` = `'whole'`, as opposed to amalgamating their formulae with the output extended Tarski formula from CAD, to create a canonical intelligible formula as would typically be produced from pure CAD.

The I'_j are not used in the genuine incremental poly-algorithm, because while lack of simplification may obfuscate the fact that they could in reality be meaningful leaves, this case is assumed to be unlikely, and the presence of quantifiers on other ineligible `IQER`s that certainly require attention means that creating a CAD only in free variables

to inspect $B^s_{j=1} I'_j$, only to later certainly extend it to one including quantified variables would be a strange choice.

Simplification of the Equivalent of the VTS Tree

One notes that *strong* simplification of (4.1) (and, more generally the intermediate equivalents of the VTS tree via (2.1) or (2.3)) may allow us to deduce a meaningful truth value for the current block of quantifiers, or otherwise a more candid representation of the quantifier free equivalent of QE. `QuantifierElimination` of course offers no such strong simplification. If it were to, one must consider that such strong simplification on the equivalent of the VTS tree, as opposed to individual formulae for `IQER`s somewhat interferes with various processes relying entirely on the tree based notions of VTS. In particular processing of VTS witnesses relies on the leaf `IQER` to act upon to be a meaningful leaf (Algorithm 3). Hence, while the state of VTS may imply a quantifier free equivalent of $x < 0 \vee x \geq 0 \equiv \mathit{true}$, where each atom owes to a different `IQER`, we will struggle to process any `IQER` to produce witnesses in this case, because the meaningful truth value for QE does not owe to a specific `IQER`. Strong simplification to a meaningful truth value may have us avoiding usage of CAD, which is good news, but if it simplifies to any other non trivial formula then we struggle to see how to use this information during or after usage of CAD in the poly-algorithm, because again we act upon individual `IQER`s at any one time. There are no such issues if strong simplification acts only on the formulae held by individual `IQER`s. For fully quantified input formulae for QE, strong simplification on individual branch `IQER`s is a matter of efficiency of solution of such `IQER`s via any methodology, rather than a matter of simplification of the end quantifier free equivalent for output, which can only be *true* or *false*. For input formulae with free variables, simplification of `IQER`s still moves the output closer to candid.

In contexts where usage of the VTS tree would certainly not be required further, strong simplification could easily be justified to deduce more candid quantifier free output from VTS. If it were deployed after every iteration of propagation of VTS (the loop of Code Fragment 4) to try to deduce early termination via meaningful truth values, one must be cognizant that the expense of simplification in this way must be balanced against the benefits (Section 2.4.2). In this case the benefits are that we may be able to terminate VTS early, but we cannot certainly expect as such.

4.4 Rich QE Output

Rich QE output generally falls to providing incrementality for QE, and witnesses for homogeneously quantified formulae. The former can be provided for poly-algorithmic QE via Section 5.1, and more broadly Chapter 5. Additionally, the latter can be provided via the canonical methodology of the poly-algorithm.

4.4.1 Production of Meaningful Witnesses for QE via VTS and CAD

Usage of the poly-algorithm for QE means that one can view a one to one correspondence between a CAD tree and exactly one `IQER`. Processing of VTS prewitnesses into witnesses involving purely real numbers requires full back substitution (Algorithm 3), and hence beginning with a meaningful leaf. Here, any ineligible `IQER` L that attributed usage of CAD is not a meaningful leaf, but the witnesses produced from CAD can provide the missing initial back substitutions when a meaningful leaf `CADCell` is found for the QE problem defined by L to emulate L being a meaningful leaf (similarly to evaluating superfluous variables at 0 before back substitution in Algorithm 3).

Fragment 45 Modifications to Algorithm 3 such that it can use witnesses from CAD in back substitution

```
3: if An Array of CAD witnesses, W, for QE on L was supplied then
4:     witnesses ← W
5:     for i from |W| + 1 to n do
6:         Append x_{n−i+1} = 0 to witnesses
7:     end for
8: else
9:     Set witnesses to be an empty Array
10:    for i from L ↦ level + 1 to n do
11:        Append x_{n−i+1} = 0 to witnesses
12:    end for
13: end if
```

Code Fragment 45 provides modification of Algorithm 3 by replacing lines 2 through 5, when Algorithm 3 also accepts an argument of W, where W are witnesses from CAD for an ineligible `IQER` L. L would no longer naturally be a meaningful leaf of the VTS tree, but manages to emulate one via this concatenation. We know that L is a meaningful leaf for $Qx_{n-m+1}, \ldots, Qx_{n-L\mapsto\text{level}}\ \Phi'(x_1, \ldots, x_{n-L\mapsto\text{level}})$, where $\Phi' = \Phi$ evaluated at W, via validity of CAD witnesses for the problem $Qx_{n-m+1}, \ldots, Qx_{n-L\mapsto\text{level}}\ L$ in the CAD solving I. Then, Algorithm 3 proceeds with producing witnesses for $Qx_{n-m+1}, \ldots, Qx_{n-L\mapsto\text{level}}\ L$, and via the concatenation, we receive witnesses for the QE problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi$.

The loop on line 5 of Code Fragment 45 exists for a similar reason to that of the loop on line 3 of Algorithm 3 — it ensures Algorithm 3 has the required "triangularity" property by ridding all expressions of free variables that were found to actually be irrelevant in light of a meaningful truth value from the CAD solving L.

As usual, it only requires one meaningful leaf `CADCell` to imply termination of QE in the conditions required for the poly-algorithm (homogeneity of quantifiers), so only one traversal from the ineligible `IQER` to the root `IQER` in Algorithm 3 concatenating with witnesses from the leaf `CADCell` is ever required to produce a set of witnesses for poly-algorithmic QE. Section 3.10 clarifies that CAD witnesses are trivial to produce, being the full sample point for a meaningful leaf `CADCell` (generated in $\mathcal{O}(n)$), so the complexity of the witness generation process for the algorithm is dominated by that of Algorithm 3.

```
> expr := exists( a, QEExamples[ `Collision` ] );
```

$$expr := (\exists a)(\exists y)(\exists x)(\exists t)\ \left(\frac{(x-t)^2}{4} + (y-10)^2 - 1 = 0\right) \wedge \frac{(-at+x)^2}{4} + (-at+y)^2 - 1 = 0 \wedge 0 < t \wedge 0 < a$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses' = false ):
> alias( R = RootOf(5*_Z^2 - 12*_Z + 6, 799503833568837/1125899906842624 .. 3198015334275375/4503599627370496) ):
> e;
```

$$[[true, a = R, x = \frac{72}{5} - 4\,R, y = \frac{49}{5}, t = \frac{2\,xa + 8\,ay - 4\,\sqrt{-a^2(x^2 - 2\,xy + y^2 - 5)}}{10\,a^2}]]$$

```
> ( e, q ) := QuantifierEliminate( expr, 'ProcessWitnesses' = true ):
> e;
```

$$[[true, a = R, x = \frac{72}{5} - 4\,R, y = \frac{49}{5}, t = \frac{1}{10\,R^2}\left(2\,R\left(\frac{72}{5} - 4\,R\right) + \frac{392\,R}{5} - \frac{8\,\sqrt{-100\,R^4 + 230\,R^3 - 101\,R^2}}{5}\right)]]$$

```
> evala( Normal( eval( GetUnquantifiedFormula( expr ), e[][ 2 .. -1 ] ) ) );
```

$$0 = 0 \wedge 0 = 0 \wedge 0 < \frac{96}{5} - 8\,RootOf\left(5\,\_Z^2 - 12\,\_Z + 6, index = 1\right) \wedge 0 < R$$

```
> map( is, % );
```

$$true \wedge true \wedge true \wedge true$$

Figure 4-2: Example of results of concatenation of CAD witnesses with those for an `IQER`, where the CAD was used to show that `IQER` was equivalent to *true*. The example 'Collision' is additionally existentially quantified by a in order to form a fully quantified formula. Presence of $\sqrt{\cdot}$ in the witness for t implies that it came purely from VTS, while presence of RootOfs in those for a and x imply they came from usage of CAD. Less obviously, the witness for a is also from CAD. While no infinitesimals need to be processed on the part of VTS, the back substitution of witnesses still provides a valid list of such. As is usually the case when RootOfs are present, usage of `evala( Normal( ... ) )` is necessary in conjunction with `is` to fully evaluate the expression at the witnesses. The keyword option `'ProcessWitnesses'` taking a boolean value controls processing of prewitnesses into witnesses by Algorithm 3 at all.

`QuantifierElimination` only ever retains the correspondence between the CAD retained in the poly-algorithm with the last `IQER` used in the poly-algorithm, and knowing that termination criteria of QE means that the meaningful leaf cell corresponds to the relevant `IQER`, because the CAD corresponds to the last used `IQER`, the concatenation implied by Code Fragment 45 correctly produces witnesses for the entire QE problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$.

In contrast, in not using the poly-algorithm (i.e. usage of the "whole" methodology for CAD by passing `'HybridMode' = 'whole'`), we cannot guarantee a one to one correspondence between the produced `CADCell`s and any one `IQER`, and so one cannot find the correct witnesses to provide to the methodology of Algorithm 3 to begin with to guarantee correct back substitution.

Chapter 5

Evolutionary Techniques

Definition 71 (Evolutionary, Incremental, Decremental). *An* Evolutionary Algorithm *refers to an incremental or decremental algorithm, as is defined below.*

An Incremental *QE algorithm is one that takes data generated alongside a solved QE problem, and produces the quantifier free output (and perhaps the modified data structures) for the previous input but with additional subformulae added within the original input formula. Such recomputation should modify existing data structures from the past computation as much as possible, hence enabling extra performance from the computation in comparison with computation of the answer from scratch.*

A Decremental *QE algorithm is as above, but with deletion of subformulae from the past QE input problem.*

Note that "evolutionary", as is defined above, is what would encompass "incremental" in most, if not all QE literature. Here, we make these three terms distinct considering "increment" has connotations of addition, and "decrement" has connotations of subtraction, and both are supported with bespoke methods here. As a specific example of comparison of terminology, "forward incremental" from [45] is roughly equivalent to "incremental" here, while "backtracking" is similarly roughly equivalent to "decremental". That work relates entirely to SMT, giving direct impetus to the term "backtracking".

Most importantly for the purposes of this work, incremental CAD completely enables the main idea behind the QE poly-algorithm discussed in Section 4. The notion of evolutionary methods are especially popular for algorithms that incur significant cost, especially when modifications to an input can appear relatively minor. Incrementality is especially useful in the context of algorithms with early termination criteria, such as deducing QE on an early clause (as is informally the intention of the methodology of SMT). Evolutionary methods accommodate replacement of a subformula from an input by another via deletion of the original subformula, followed immediately by an insertion. The sentiment of evolutionary methods has us attempting to avoid the more costly operations in both VTS and CAD alike, such as:

- factorisation of polynomials within relations to deduce if an `IQER` is ineligible, or

for structural test point generation (at-cs-fac),

- virtual substitution of structural test points into formulae in general, which in general reduces to pseudoremainders (vs-prd-at),
- the polynomial operations associated with projection (projectAllOrders), and the associated generation of bases which involves factoring,
- substitution of real algebraic numbers into constraints in CAD (evaluateTFArrayAtSP),
- and real root isolation in order to lift new cells (CCHILD).

The main aspect of incrementality in CAD is addition of new projection polynomials owing to new constraints, and the associated new lifting due to new roots of associated new lifting polynomials at any one cell. The main aspect of incrementality in VTS is generation and usage of new structural test points owing to new constraints. Incrementality in CAD should largely revolve around addition of projection polynomials and any associated lifting as a result. Incrementality in VTS should revolve around addition of constraints, and any associated structural test points.

This section hopes to provide new methods and explanation behind evolutionary QE, including for the poly-algorithmic system. Evolutionary QE may be desirable in the context of SMT, where SMT iteratively adds new constraints for the theory solver (in this case, a QE implementation) to solve. This corresponds to incrementality, but furthermore decrementality enables "backtracking" as is commonly used in SMT. One notes that full incrementality to achieve QE on any formula has the disadvantage against the non incremental case that not all information is available at the first time of asking to make decisions on various aspects of strategy such as variable strategy — such a choice may be good early, but not later on in light of new constraints. The cost of incrementality in this sense in is intended to be offset by early termination without usage of all constraints, as is the case in SMT or for `QuantifierElimination`'s poly-algorithm.

In many other cases for incrementality, all that is required is forming a new disjunction or conjunction of a new formula with the entire formula used in the past computation. Here, a more general framework is presented that allows for new formulae to be created at an arbitrary position with arbitrary operator within an existing formula. The hope is that such general evolutionary techniques enable even inexperienced users to explore QE problems in more depth. The formulation of "theorems" in QE [54] is especially relevant here, and Section 5.1.1 explores case studies demonstrating when evolutionary methods acting at arbitrary "atomic positions" can be useful to explore problems in detail, sans the expense of constantly calling QE from scratch.

One notes existing approaches to incrementality for VTS or CAD in QE and/or SMT, such as [45, 17, 19]. The notion of atomic position (Section 5.1.1) here is new, allowing for more generic evolutionary methods, and more attention is paid to the distinction between *incrementality* and *decrementality*. Otherwise, the methods are more bespoke given the context of the operation of `QuantifierElimination`, such as giving rise to poly-algorithmic evolutionary methods, and the need to pay attention to

the facets of using a Lazard projection CAD with ECs, such as curtains. In fact, some of the evolutionary methods used here enable the realisation of methods to recover from certain Lazard curtains.

Due to `QuantifierElimination` being largely object oriented via `IQER`s, `CADCells`, and `projection` objects, one notes that most of the incremental functions to achieve incrementality, especially those concerned with tree traversal, are object methods in practice.

The functions InsertFormula and DeleteFormula in `QuantifierElimination` implement incrementality and decrementality with the semantics described by Algorithms 47 and Algorithms 54 for incrementality and Algorithms 49 and Algorithms 56 and decrementality. InsertFormula and DeleteFormula use an appropriate algorithm for poly-algorithmic QE or pure Partial CAD depending on whether they are passed `QEData` or `CADData`. In particular, InsertFormula and DeleteFormula work in terms of "atomic position" (Definition 72), and for InsertFormula one must also hence supply the name of a boolean operator.

5.1 Evolutionary VTS & Poly-algorithmic QE

Considering the canonical tree structure formed by VTS on a block of quantifiers, with `IQER`s storing as much information as we require to enable further computation, most evolutionary VTS requires traversal of the VTS tree to "fix up" existing information in the tree such that it reflects the new problem after insertion or deletion of a formula. The formulae of the nodes of the VTS tree will then be semantically correct in light of the new implied input for QE, but the resulting tree may not necessarily be sufficient to describe a quantifier free answer for the new problem. This traversal must insert or delete (for incrementality or decrementality respectively) a given subformula as prescribed by the user from each node, which associates a quantifier free formula via the result of a sequence of virtual substitutions. In the case of insertion, one need only insert the new formula modulo the test points used to acquire this node. Via the object based implementation, each node stores the test point used to obtain it from the `IQER` above (essentially the preceding edge).

Because the evolutionary aspects of VTS act on a single VTS tree, evolutionary VTS is only supported for homogeneously quantified QE problems. Requesting `QEData` for a problem not homogeneously quantified for non homogeneously quantified QE problems results in an error in `QuantifierElimination`, however evolutionary CAD does support this case.

Most of the concepts for Section 5.1.2 and 5.1.3 were first presented in [63]. In particular Algorithms 46 and 48 come from that work, but here are canonicalized in terms of `IQER`s rather than generic "VTS nodes", and there is some extra functionality to enable poly-algorithmic incrementality on ineligible `IQER`s via functions inherited from the poly-algorithmic method (Section 4).

So far we have talked of insertion or deletion of formulae, but have no meaningful concepts to make this well defined.

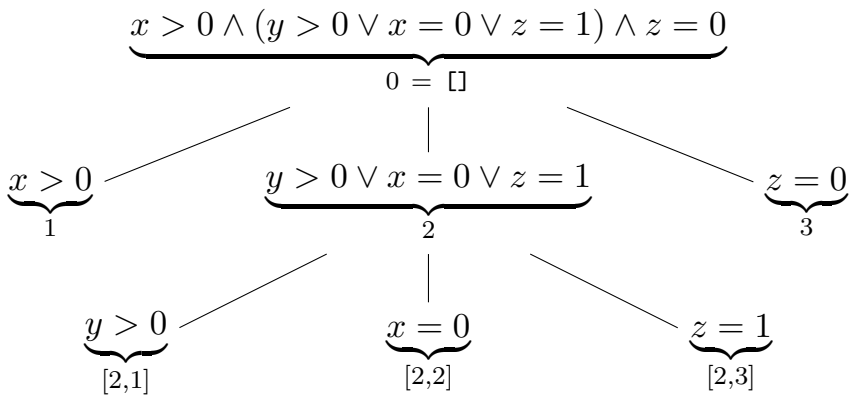


Figure 5-1: Demonstration of all valid atomic positions on the formula $x > 0 \wedge (y > 0 \vee x = 0 \vee z = 1) \wedge z = 0$. The expression under a brace shows the atomic position of the given subformula.

5.1.1 Structural Tarski Formulae & Atomic Position for Evolutionary QE

The notion of atomic position was presented in [63].

Definition 72 (Atomic Position). *The* Atomic Position α *of some subformula of a formula* Φ *is its index when viewed as an operand of* Φ *when* Φ *has outer operator* $\wedge$ *or* $\vee$*, or the concatenation of the index of the operand it is contained in, say* i*, with its atomic position within operand* i*, when operand* i *is a conjunction or disjunction. If we wish to speak of the atomic position of* Φ *itself, we refer to this as atomic position 0. When referring to an atomic position for insertion of a new formula* Ψ *within* Φ*, one must specify an operator (*`And/Or`*) such that the subformula at atomic position* α *either forms a conjunction or disjunction respectively with the new formula* Ψ *to insert.*

Figure 5-1 exemplifies atomic position on a formula featuring nesting. The concatenation of atomic positions to describe the position of a nested formula is semantically achieved by a flat Maple list, delimited by square brackets. The atomic position 0 is equivalent to the empty list, `[]`. Atomic positions corresponding to a single integer are also equivalent to a list delimiting the same integer. The overall notion is very similar to that of "position" in [43, Section 3], where the intention there is to achieve a feature in test point generation called "prime constituents". Here, the view is entirely towards achieving general incrementality, including with respect to CAD. Hence the definition here also alludes to boolean operators such that one can extend or modify a particular subformula. Another menial difference is that the atomic position of the "root" formula can be referred to as 0 here.

Both VTS and CAD use structural (real) Tarski formulae in order to enable general evolutionary methods that can act at general atomic positions within a formula. The idea of structural Tarski formulae is that they are weak simplified without the typical short circuiting of $\wedge$ and $\vee$ by *false* and *true* respectively. That is, the formula $x < 0 \wedge 0 < 0$ simplifies to $x < 0 \wedge$ *false* rather than *false* under weak simplification preserving structural form. In this way, we can retain a one to one correspondence

between the results of (virtual) substitution on a specific atom within the formula for an `IQER` or `CADCell` and the same atom in the top level unquantified formula for QE, Φ. As discussed in Section 2.4.2, this correspondence lends the context for "local" in the locally canonical representations of the structural formulae within such objects. More generally, we can retain a correspondence between a specific subformula from input and the results of a sequence of virtual or algebraic substitutions from VTS or CAD respectively on that exact subformula. Both `IQER`s and `CADCell`s still retain a weak simplified non structural formula via properties formulaSimplified and tarski_formula respectively. The properties storing the structural formulae are structuralSubstitution and tarski_formula_structural. Structural substitution uses the facts:

- `IQER`s and `CADCells` both inherit parenting as nodes within respective trees for VTS and CAD. When they are tree nodes of a positive level, one (perhaps virtually) substitutes a term into the formula held by its parent node to receive its own formula. For `IQER`s this depends on the test point having a non trivial guard, and for `CADCell`s this happens in evaluation of the cell.

- There is no reason to reorder operands within any formula under (virtual) substitution. For example, the algebraic substitution of a rational number a for x into $f(x) > 0 \wedge g(x) = 0$, $(f(x) > 0 \wedge g(x) = 0)[x \,/\, a] \equiv (f(x) > 0)[x \,/\, a] \wedge (g(x) = 0)[x \,/\, a]$ after distribution of the substitution, and there is no reason to reorder the operands such that the formula is now represented as $(g(x) > 0)[x \,/\, a] \wedge (f(x) > 0)[x \,/\, a]$, ruining the correspondence between atoms under substitution that we need to preserve in order to realise atomic position.

- For CAD, standard substitution of real algebraic numbers trivially maps atoms to atoms. For VTS, virtual substitution can map atoms to any quantifier free Tarski formula, i.e. the atom may "expand" under virtual substitution, *but* structural substitution allows for nesting within formulae such that a conjunction can be a genuine operand below a conjunction, such that the correspondence enabled by atomic position is preserved.

- The structural formulae within `IQER`s are stored away from the associated guard for the `IQER`, because the guard is an "appendage" to the structure of the original formula Φ after successive virtual substitutions, e.g. by conjunction with the results of virtual substitution in the existential case, and so storing the guard in structural form is not only superfluous (considering the intention is never to act on the guard for any `IQER`), but destroys the notion of atomic position on `IQER`s. However, the simplified formula held by an `IQER` still reflects the formula including the guard.

As a case study as to why one may be interested in general evolutionary methods via atomic positions, consider QE to prove a "theorem" $\forall \mathbf{x}\ A(\mathbf{x}) \Rightarrow H(\mathbf{x})$ where $A(x)$ is a formula of "assumptions" in conjunctive normal form (CNF), and $H(x)$ a formula of "hypotheses" to prove, also in CNF. This "theorem" formulation for QE is discussed in [54]. After conversion to prenex form this formula is $\forall \mathbf{x}\ \neg A(\mathbf{x}) \vee H(\mathbf{x})$, so $\neg A(\mathbf{x})$ is now in disjunctive normal form. Atomic position enables modification of operands

within any of the disjunctions below the conjunction that is $H(\mathbf{x})$, by deletion of a subformula at a particular atomic position, and then placement of another.

As a small and easy to understand example, consider the example $\forall x\ (x < 0 \vee x = 0) \Rightarrow x^2 > 0$, which after conversion to prenex form becomes

$$\forall x\ (x \geq 0 \wedge x \neq 0) \vee x^2 > 0.$$

One must examine atomic positions of the prenex formula, because `QuantifierElimination` only works in terms of prenex formulae. The formula is of course equivalent to *false* via QE using any methodology, with $x = 0$ actually being the falsifying witness. In order to investigate changing the constraint $x \neq 0$ to $x > 1$, corresponding to exchanging the "assumption" $x < 0 \vee x = 0$ in the non prenex theorem with $x < 0 \vee x \leq 1$, one can delete the operand at atomic position $[1, 2]$ (second operand of the first operand of the unquantified formula), such that the represented formula is now

$$\forall x\ x \geq 0 \vee x^2 > 0$$

($\forall x\ x < 0 \Rightarrow x^2 > 0$ imitating the non prenex structure), which evolutionary QE finds to be *true*. The atom $x \geq 0$ is now at atomic position 1, because structural form allows for *true* or *false* operands, but not usage of `And` or `Or` as unary operators, so the `Or` has collapsed. One can then insert the constraint $x > 1$ at atomic position 1 with operation `And` such that the QE data now represents

$$\forall x\ (x \geq 0 \wedge x < 1) \vee x^2 > 0$$

(equivalent to the non prenex $\forall x\ (x < 0 \vee x \geq 1) \Rightarrow x^2 > 0$ imitating the original structure) with QE deducing this formula is *true*. The `Or` has once again been reinstated via this insertion with operator.

More generally, the hope is that one can experiment with the formulation of "theorems" via evolutionary methods via atomic positions in order to better understand problems in QE, while making the process as efficient as possible by retention of data structures as is the usual aim of incrementality. This also allows for the sentiment of [54] for examining economics theorems of first performing the cheaper check of the compatibility of the "assumptions" before extending the result to checking for existence of an example by addition of the hypotheses.

Atomic position still realises the usual notion of incrementality in QE for uses such as SMT. For a formula $\bigwedge_{i=1}^{k} \Psi_i$ in CNF such that each Ψ_i is a disjunction of constraints, addition of a constraint or new disjunction Ψ_{k+1} can be achieved by specifying any integer $0 \leq i \leq k + 1$ and operator `And`, such that the QE problem reflects $\bigwedge_{i=1}^{k+1} \Psi_i$ after incrementality. Of course, this insertion avoids nesting such that the formula is a flat conjunction of disjunctions, rather than a nested conjunction within a conjunction, and the structural formulae held by `IQER`s and `CADCell`s are stored as Arrays such that they are mutable in order to support insertion.

Note that the evolutionary methods offered by `QuantifierElimination` do not enable deletion or addition of quantifiers.

5.1.2 Incremental VTS & Poly-algorithmic QE

We describe two algorithms in turn to describe the case for incremental poly-algorithmic QE (which for the most part consists of incremental VTS). Note that if the formula to insert ψ features irrational numbers (i.e. real algebraic numbers that are RootOfs), VTS is inapplicable, and therefore evolutionary QE on `QEData` in `QuantifierElimination` via InsertFormula produces an error if a Real Tarski formula is passed. While addition of radicals for VTS could be accommodated by replacement of $\sqrt[d]{k}$ by x and addition of the clause $\exists x\ x^d = k$ (only when $d \leq 2$ could this constraint be processed by VTS) to an outer conjunction encapsulating the original formula, this corresponds to addition of quantifiers, which we have already excluded. Such replacements are not generally discussed in this work otherwise. Hence ψ in Algorithms 46 and 47 is required to be a genuine Tarski formula (Definition 3).

Algorithm 46 is a recursive tree traversal algorithm to "fix" the VTS tree in terms of a new formula ψ to insert at atomic position α. The tree traversal occurs root down. We insert ψ at the appropriate atomic position into the structural formula for the `IQER` to act upon. This is done modulo the virtual substitutions defined by the path to this `IQER`, via recursion. We must also take into account the quantifier for the overarching block of variables for this VTS tree. Upon simplifying the structural formula after insertion, the `IQER`'s simplified formula may be equivalent to *true* or *false*, and we can act accordingly to truncate the VTS tree, or notify that the `IQER` is now a "work `IQER`" amenable for further QE (it is not identified as ineligible here). Otherwise we perform set arithmetic to deduce the contribution of any *new* test points from ψ.

Proposition 73. *Usage of Algorithm 46 modifies the VTS subtree rooted at I such that it reflects a correct VTS tree for the QE problem represented by I after insertion of ψ at atomic position α with operation* oper *into the formula of I (via its structural formula).*

Proof.

1. The new formula to insert, ψ, is inserted at each `IQER` modulo the appropriate sequence of virtual substitutions defined by the path from self to the root `IQER`. This is because we recurse with S, which is the result of virtual substitution on ψ. This is modulo the overarching quantifier, i.e. if $Q = \forall$, then we need to "negate the virtual substitution of negation" of ψ. Structural formulae do not contain the guards for each `IQER`, so after insertion of the substituted term S into the structural formula for I, we simplify it to an unstructural formula, and build the appropriate full simplified formula for I including via its guard $I \mapsto$ guard.

2. Insertion trivially corresponds to addition of information. Therefore, tree pruning as can be applicable for deletion (Section 5.1.4) is inapplicable.

3. I is essentially treated in terms of its new simplified formula (which could easily coincide with its previous simplified formula). If I is a meaningful leaf after insertion, any subtree it previously held is now defunct, and hence discarded. If I was previously a meaningful leaf, but its formula after non structural simplification is not *true* or *false*, then it is added to the container iqers such that it now receives further QE depending on if it is ineligible or not (at the top level).

Algorithm 46 Recursive tree traversal to insert a Tarski formula into the formula for an `IQER` at a certain atomic position

Input: I an `IQER`, iqers a container of non genuine leaf `IQER`s, leaves a container for genuine leaf `IQER`s, Q the quantifier symbol for the current block of quantifiers, vars the `Array` of quantified variables from the current block $[x_{n-m+1}, \ldots, x_n]$, α an atomic position, oper a boolean operator, ψ the Tarski formula to insert

Output: No meaningful return — modifies the VTS subtree rooted at I inplace

1: **procedure** TRAVERSEVTSTREEINSERT(I, iqers, leaves, Q, vars, α, oper, ψ)
2: **if** $I \mapsto$ level $= 0$ **then**
3: Insert ψ at atomic position α into $I \mapsto$ formulaSimplified
4: $I \mapsto$ formulaSimplified $\leftarrow$ simplify($I \mapsto$ formulaSimplified)
5: **if** $I \mapsto$ formulaSimplified $=$ *true* or $I \mapsto$ formulaSimplified $=$ *false* **then**
6: Add I to leaves
7: **elseif** $I \mapsto$ cad_formula is an (extended) Tarski formula **then** ▷ CAD was used on this `IQER` before
8: Unassign $I \mapsto$ cad_formula
9: Add I to iqers
10: **elseif** I has children `IQER`s ($I \mapsto$ children) **then**
11: **if** $Q = \exists$ **then**
12: $\psi' \leftarrow \psi$
13: **else** ▷ $Q = \forall$
14: $\psi' \leftarrow \neg\psi$
15: **end if**
16: $E \leftarrow$ PC-to-TPs(atposl(ψ', x_n), x_n)\{$c \mapsto$ testpoint $\mid c \in I \mapsto$ children}
17: $I \mapsto$ futureTestpoints $\leftarrow I \mapsto$ futureTestpoints $\cup E$
18: **if** $|I \mapsto$ futureTestpoints$| > 0$ **then**
19: Add I to iqers
20: **end if**
21: **for** c in $I \mapsto$ children **do**
22: traverseVTSTreeInsert(c, iqers, leaves, Q, vars, α, oper, ψ)
23: **end for**
24: **else**
25: Add I to iqers
26: **end if**
27: **else**
28: **if** $Q = \exists$ **then**
29: $S \leftarrow \psi[x_{n-I\mapsto\text{level}} \,//\, I \mapsto$ testpoint], simplifying weakly but preserving structural form
30: Insert S at atomic position α with operation oper in $I \mapsto$ structuralSubstitution
31: $I \mapsto$ formulaSimplified $\leftarrow I \mapsto$ guardFormula$\wedge$simplify($I \mapsto$ structuralSubstitution)

Algorithm 46 Recursive tree traversal for VTS insertion, Part 2

32: **else** ▷ $Q = \forall$
33: $S \leftarrow \neg((\neg\psi)[x_{n-I\mapsto\text{level}} \,//\, I \mapsto \text{testpoint}])$, simplifying weakly but preserving structural form
34: Insert S at atomic position α with operation oper in $I \mapsto$ structuralSubstitution
35: $I \mapsto$ formulaSimplified $\leftarrow I \mapsto$ guardFormula$\vee$ simplify($I \mapsto$ structuralSubstitution)
36: **end if**
37: **if** $I \mapsto$ formulaSimplified $=$ *true* or $I \mapsto$ formulaSimplified $=$ *false* **then**
38: **if** $I \mapsto$ formulaSimplified is a meaningful truth value for Q **then**
39: Add I to leaves
40: **end if**
41: Unassign I's cad_formula, futureTestpoints, children, hence discarding the VTS subtree rooted at I
42: **elseif** $I \mapsto \text{level} = m$ **then**
43: Add I to leaves
44: **elseif** $I \mapsto$ cad_formula is an (extended) Tarski formula **then** ▷ CAD was used on this `IQER` before
45: Unassign $I \mapsto$ cad_formula
46: Add I to iqers
47: **elseif** I has children `IQER`s ($I \mapsto$ children) **then**
48: **if** $Q = \exists$ **then**
49: $S' \leftarrow S$
50: **else** ▷ $Q = \forall$
51: $S' \leftarrow \neg S$
52: **end if**
53: $E \leftarrow$ PC-to-TPs(atposl(S', $x_{n-I\mapsto\text{level}-1}$), $x_{n-I\mapsto\text{level}-1}$) $\setminus\{c \mapsto \text{testpoint} \mid c \in I \mapsto \text{children}\}$
54: $I \mapsto$ futureTestpoints $\leftarrow I \mapsto$ futureTestpoints $\cup E$
55: **if** $|I \mapsto \text{futureTestpoints}| > 0$ **then**
56: Add I to iqers
57: **end if**
58: **for** c in $I \mapsto$ children **do**
59: traverseVTSTreeInsert(c, iqers, leaves, Q, vars, α, oper, S)
60: **end for**
61: **elseif** $I \mapsto$ futureTestpoints is not a set (i.e. not computed) or $|I \mapsto \text{futureTestpoints}| > 0$ **then**
62: Add I to iqers
63: **end if**
64: **end if**
65: **return**
66: **end procedure**

4. An `IQER` was certainly previously ineligible with respect to the last pass of QE if it attributes a cad_formula (lines 7 and 44). It may no longer be ineligible after insertion, say if S is *true*, and its insertion into a disjunction contained within the formula for I now short circuits that disjunction which previously contained the one (or more) constraint of excessive degree making I ineligible. The `QEData` from the last pass of QE always retains `CADData`. That being said, it is undesirable to immediately attempt to repurpose the CAD for an `IQER` in this scenario for two reasons. First, we would like to retain correspondence between the held CAD and any `IQER` below which we found a meaningful leaf cell, such that we can necessarily yield poly-algorithmic witnesses (Section 4.4.1). Repurposing all ineligible `IQER`s in the tree doesn't allow us to do this, as traversal of the algorithm traverses root down irregardless of finding meaningful truth values or otherwise, as it has no knowledge of what is happening "in other parts of the tree". Secondly, immediately deploying some usage of CAD on such `IQER`s doesn't fit with our usual mantra of "do as much VTS as possible, then proceed with poly-algorithmic QE if necessary". We may even find a meaningful leaf `IQER` in this tree traversal via insertion that we do not have prior knowledge of yet due to the recursive methodology, and in this case usage of CAD would largely be "for the sake of it". As a result, hitting any `IQER` that was previously ineligible has us adding the `IQER` back to the container of "work" `IQER`s, iqers, and the top level calling function has the responsibility of solving it with appropriate methodology depending on if it is still ineligible. Either way, doing this allows us to be consistent with poly-algorithmic methodology. In any case, the existing cad_formula is potentially erroneous in light of the inserted sub-formula, so must unfortunately be discarded until the `IQER` can otherwise attribute a quantifier free equivalent.

5. Lines 16 and 17, or 53 and 54 reflect the set arithmetic to generate any new unique test points owing only to the inserting formula. The test points already generated and/or used on I exist in the set $I \mapsto$ futureTestpoints, or via the branches below I. We merge in the new test points to the $I \mapsto$ futureTestpoints property, and if this yields a non empty set, I is amenable to further propagation of VTS. The generation of test points takes into consideration the overarching quantifier Q, i.e. we may need to generate test points from the negation.

6. All temporary leaves and genuine leaves of the VTS subtree rooted at I are added to iqers and leaves respectively.

7. Lack of prime constituents from [43, Section 3.1] makes lines 16 and 53 simpler than they may or would be in the presence of such an implementation with prime constituents — the test point sets to use here are always "flat" in that they use all atoms of the formula equally.

Proposition 74. *Usage of Algorithm 47 QEIncremental achieves the goal of incremental poly-algorithmic QE, enabling insertion of a new formula ψ into Φ at atomic position*

Algorithm 47 Incremental poly-algorithmic QE, via insertion of a new formula at a certain atomic position

Input: data, `QEData` for a previous homogeneously quantified problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$, α an atomic position, ψ an unquantified Tarski formula to insert at atomic position α with operation "oper" in Φ, "oper" a boolean operator to insert with

Output: QE output dependent on number of output arguments requested, up to and including the quantifier free equivalent, witnesses, and the `QEData`

1: **procedure** QEIncremental(data, α, ψ, oper)
2: iqers $\leftarrow$ an empty container
3: leaves $\leftarrow$ an empty container
4: Let data $\mapsto$ RootIQER be the root `IQER` for the past QE computation, data $\mapsto$ Vars the `Array` of variables for the past computation, and C = data $\mapsto$ cad_data any retained `CADData` from the past computation for poly-algorithmic QE
5: traverseVTSTreeInsert(data $\mapsto$ RootIQER, iqers, leaves, Q, data $\mapsto$ Vars, α, oper, ψ)
6: **if** There exists an `IQER` in leaves with a meaningful truth value for Q **then**
7: iqers $\leftarrow$ an empty container
8: **end if**
9: Code Fragment 4 ▷ Propagation of VTS on non ineligible `IQER`s on the container iqers
10: Code Fragment 42 ▷ Poly-algorithmic QE on ineligible `IQER`s
11: **return** QE output dependent on number of output arguments requested
12: **end procedure**

α with operation oper *for a past homogeneously quantified QE problem* $Qx_{n-m+1}, \ldots, Qx_n\ \Phi$ *donating* `QEData` data *from that elimination, such that we can produce QE output for the problem after insertion.*

Proof. Via Proposition 73, calling Algorithm 46 at the root `IQER` modifies the existing VTS tree to reflect the problem after insertion. In particular, with iqers and leaves defined as empty containers beforehand, they are now a container of all non genuine leaf `IQER`s and genuine leaf `IQER`s. While the tree is now correct, it may not be *sufficient* to describe a quantifier free formula for the new QE problem in incrementality, hence if no meaningful truth values were yielded from the tree traversal (but non genuine leaf `IQER`s were), then regular QE by propagation of VTS on non ineligible `IQER`s into poly-algorithmic QE on any ineligible `IQER`s continues, and use of such is much the same as the non evolutionary case. Retention of the CAD, if one was produced at all, from the last usage of poly-algorithmic QE for the overarching problem, enables full incrementality for poly-algorithmic QE where it would be unclear in a less bespoke VTS into CAD approach.

5.1.3 Decremental VTS & Poly-algorithmic QE

Next we describe the case for decremental poly-algorithmic QE (which mostly consists of delineation of decremental VTS).

Algorithm 48 is a recursive tree traversal algorithm to "fix" the VTS tree in terms of deletion of a subformula at a particular atomic position from an originally traversed QE problem. By recursive traversal of the tree root down, we can delete the subformula of each `IQER` from its structural formula which corresponds precisely to a sequence of virtual substitutions on the same subformula from the root `IQER`. Much of the logic of the algorithm inspects the new state of the `IQER` in terms of the resulting simplified formula. We have a choice to prune VTS subtrees of test points no longer attributable to any existing atom, via some set arithmetic on generation of test points.

Proposition 75. *Usage of Algorithm 48 modifies the VTS subtree rooted at I such that it reflects a correct VTS tree for the QE problem represented by I after deletion of the formula at atomic position α from the unquantified formula for I, and then recursion of the subtree beneath I.*

Proof.

1. The structure of Algorithm 48 is much the same as Algorithm 46, except that we perform deletion instead of insertion on the structural formulae associated with `IQER`s through traversal on the VTS tree. Much of the logic on the treatment of the `IQER` is therefore the same, and we move to discuss the unique points.

2. An `IQER` I associating a cad_formula can more easily be seen to no longer be ineligible than the case for insertion if we delete any subformula from I's structural substitution that contains all the constraints of excessive degree. Much like in Algorithm 46, we do not even attempt to deduce if it is ineligible or not — we must discard the cad_formula, which is now potentially erroneous in light of deletion, and adding I to the container of "work `IQER`s" iqers has that the top level handling deletion can proceed with QE on I in whatever way is appropriate, whether it is ineligible now or not.

Algorithm 48 Recursive tree traversal to delete a subformula from an `IQER` at a certain atomic position

Input: I an `IQER`, iqers a container of non leaf `IQER`s, leaves a container for leaf `IQER`s, Q the overarching quantifier symbol for the block of quantifiers for VTS, vars the `Array` of quantified variables fro the current block $[x_{n-m+1}, \ldots, x_n]$, α an atomic position, and PruneTree, a boolean flag dictating whether the user wishes to "prune" the VTS tree

Output: No meaningful return — modifies the VTS subtree rooted at I inplace

1: **procedure** traverseVTSTreeDelete(I, iqers, leaves, Q, vars, α, PruneTree)
2: **if** $I \mapsto$ level $= 0$ **then**
3: Delete the subformula at atomic position α from $I \mapsto$ formulaSimplified. Let ψ be the formula that was removed.
4: $I \mapsto$ formulaSimplified $\leftarrow$ simplify($I \mapsto$ formulaSimplified)
5: **if** $I \mapsto$ formulaSimplified $=$ *true* or $I \mapsto$ formulaSimplified $=$ *false* **then**
6: **if** $I \mapsto$ formulaSimplified is a meaningful truth value for Q **then**
7: Add I to leaves
8: **end if**
9: Unassign I's cad_formula, futureTestpoints, children, hence discarding the VTS subtree rooted at I
10: **elseif** $I \mapsto$ cad_formula is an (extended) Tarski formula **then** ▷ CAD was used on this `IQER` before
11: Unassign $I \mapsto$ cad_formula
12: Add I to iqers
13: **elseif** I has children `IQER`s ($I \mapsto$ children) **then**
14: **if** PruneTree **then**
15: **if** $Q = \exists$ **then**
16: $E \leftarrow$ PC-to-TPs(atposl(ψ, x_n), x_n) $\setminus$ PC-to-TPs(atposl($I \mapsto$ formulaSimplified, x_n), x_n)
17: **else** ▷ $Q = \forall$
18: $E \leftarrow$ PC-to-TPs(atposl($\neg\psi$, x_n), x_n) $\setminus$ PC-to-TPs(atposl($\neg($ $I \mapsto$ formulaSimplified), x_n), x_n)
19: **end if**
20: $I \mapsto$ futureTestpoints $\leftarrow I \mapsto$ futureTestpoints $\setminus E$
21: **for** c in $I \mapsto$ children **do**
22: **if** $c \mapsto$ testpoint $\in E$ **then**
23: Remove c from $I \mapsto$ children
24: **else**
25: traverseVTSTreeDelete(c, iqers, leaves, Q, vars, α, PruneTree)
26: **end if**
27: **end for**
28: **else**
29: **for** c in $I \mapsto$ children **do**
30: traverseVTSTreeDelete(c, iqers, leaves, Q, vars, α, PruneTree)
31: **end for**
32: **end if**

Algorithm 48 Recursive tree traversal to delete a subformula from an `IQER`, Part 2

```
33:         elseif I ↦ futureTestpoints is not a set (i.e. not computed) or
                |I ↦ futureTestpoints| > 0 then
34:             Add I to iqers
35:         end if
36:     else
37:         Delete the subformula at atomic position α from
                I ↦ structuralSubstitution. Let ψ be the formula that was removed.
38:         if Q = ∃ then
39:             I ↦ formulaSimplified ← I ↦ guardFormula∧ simplify(
                    I ↦ structuralSubstitution )
40:         else
41:             I ↦ formulaSimplified ← I ↦ guardFormula∨ simplify(
                    I ↦ structuralSubstitution )
42:         end if
43:         if I ↦ formulaSimplified = true or I ↦ formulaSimplified = false then
44:             Remove the whole VTS tree from iqers (by removing everything from
                    iqers)
45:             Add I to leaves
46:         elseif I ↦ level = m then
47:             Add I to leaves
48:         elseif I ↦ cad_formula is an (extended) Tarski formula then   ▷ CAD was
                used on this IQER before
49:             Unassign I ↦ cad_formula
50:             Add I to iqers
51:         elseif I has children IQERs (I ↦ children) then
52:             if PruneTree then
53:                 if Q = ∃ then
54:                     E ← PC-to-TPs( atposl( ψ, x_{n−I↦level} ), x_{n−I↦level} ) \
                            PC-to-TPs( atposl( I ↦ formulaSimplified, x_{n−I↦level} ),
                            x_{n−I↦level} )
55:                 else                                                  ▷ Q = ∀
56:                     E ← PC-to-TPs( atposl( ¬ψ ,x_{n−I↦level} ), x_{n−I↦level} ) \
                            PC-to-TPs( atposl( ¬(I ↦ formulaSimplified ), x_{n−I↦level} ),
                            x_{n−I↦level} )
57:                 end if
58:                 I ↦ futureTestpoints ← I ↦ futureTestpoints \ E
59:                 for c in I ↦ children do
60:                     if c ↦ testpoint ∈ E then
61:                         Remove c from I ↦ children
62:                     else
63:                         traverseVTSTreeDelete( c, iqers, leaves, Q, vars, α,
                                PruneTree )
64:                     end if
65:                 end for
66:             else
```

Algorithm 48 Recursive tree traversal to delete a subformula from an `IQER`, Part 3

```
67:                 for c in I ↦ children do
68:                     traverseVTSTreeDelete( c, iqers, leaves, Q, vars, α, PruneTree )
69:                 end for
70:             end if
71:         elseif I ↦ futureTestpoints is not a set (i.e. not computed) or
                |I ↦ futureTestpoints| > 0 then
72:             Add I to iqers
73:         end if
74:     end if
75:     return
76: end procedure
```

3. Because deletion of subformulae corresponds to removal of information, one can consider "pruning" the VTS tree of subtrees that owe from now redundant substitutions. One deduces the redundant information by generating the set of structural test points for the subformula that was deleted, and taking the set difference of that with the new simplified formula for I resulting from deletion. The set of test points E deduced to be redundant means that we can not only delete subtrees immediately owing to test points in E for those children which have a test point $T \in E$, but take the set difference of the set of future test points for elimination on I, $I \mapsto$ futureTestpoints with E to ensure they are not used. A short cost benefit analysis of doing such a thing is discussed in Section 5.1.4. Pruning the tree is controlled by the keyword option boolean flag `'PruneTree'` in `QuantifierElimination` that appears as an argument to Algorithm 48.

4. Much as the case for insertion, lack of prime constituents from [43, Section 3.1] makes the test point generation in pruning "easy" in the sense that we know test point generation is "flat", with contribution from all atoms of a formula equally. If the implementation were to take this into account, this pruning would likely become more convoluted.

Proposition 76. *Usage of Algorithm 49 QEDecremental achieves the goal of decremental poly-algorithmic QE, enabling deletion of a subformula at atomic position α from Φ for a past homogeneously quantified QE problem $Qx_{n-m+1}, \ldots, Qx_n$ Φ donating* `QEData` data *from that elimination, such that we can produce QE output for the problem after this deletion.*

Proof. The case is much the same as for QEIncremental. In traversal of the whole VTS tree by Algorithm 48, the VTS tree now represents a correct tree for VTS on Φ after deletion. However, again, the tree may not be *sufficient* to describe quantifier free output, for example if all `IQER`s that were previously meaningful leaves are now non genuine leaves due to the deletions. In the canonical containers for `IQER`s, iqers and leaves being set as empty containers before passing to the tree traversal, they are now

Algorithm 49 Decremental poly-algorithmic QE, via deletion of a subformula at a certain atomic position

Input: data `QEData` for a previous homogeneously quantified problem $Qx_{n-m+1}, \ldots, Qx_n\ \Phi(x_1, \ldots, x_n)$, α the atomic position of the subformula the user wishes to delete from computation, and a boolean flag "PruneTree" representing if the user wishes to prune the VTS tree of defunct `IQER`s

Output: QE output dependent on number of output arguments requested, up to and including the quantifier free equivalent, witnesses, and the `QEData`

1: **procedure** QEDecremental(data, α, PruneTree)
2: iqers $\leftarrow$ an empty container
3: leaves $\leftarrow$ an empty container
4: Let data $\mapsto$ RootIQER be the root `IQER` for the past QE computation, data $\mapsto$ Vars the `Array` of variables for the past computation, and C = data $\mapsto$ cad_data any retained `CADData` from the past computation for poly-algorithmic QE
5: traverseVTSTreeDelete(data $\mapsto$ RootIQER, iqers, leaves, Q, data $\mapsto$ Vars, α, PruneTree)
6: **if** There exists an `IQER` in leaves with a meaningful truth value for Q **then**
7: iqers $\leftarrow$ an empty container
8: **end if**
9: Code Fragment 4 ▷ Propagation of VTS on non ineligible `IQER`s on the container iqers
10: Code Fragment 42 ▷ Poly-algorithmic QE on ineligible `IQER`s
11: **return** QE output dependent on number of output arguments requested
12: **end procedure**

populated with non genuine leaf IQERs and genuine leaf IQERs respectively. Therefore, if we have not received a meaningful truth value from any IQER as a result of tree traversal, we proceed with standard propagation of VTS, followed by poly-algorithmic QE on ineligible IQERs if necessary to generate quantifier free output. Retention of the CAD, if one was produced at all, from the last usage of poly-algorithmic QE for the overarching problem enables full decrementality for poly-algorithmic QE where it would be more difficult in a less bespoke VTS into CAD approach.

5.1.4 VTS Tree Pruning

Usage of too many test points in VTS is never incorrect — the only requirement for VTS to provide the correct answer is that *sufficiently many* of test points are used to cover the real line corresponding to a quantified variable, and as such it suffices to substitute one point from every interval implied by real roots of polynomials in that quantified variable. Using the fewest is a matter of efficiency to avoid the act of virtual substitution, which incurs cost in itself. Having performed VTS on an input formula (1.1), and then using the above framework to delete a subformula of Φ, means that existing test points used in the VTS tree may no longer be able to be attributed to any relation from the IQER above it after deletion. Hence, we may obtain a surplus of test points beneath certain IQERs. Given the usage of test points in this case is sufficient but not necessary in terms of the quantifier free equivalent to eventually receive, one may wish to remove these test points, which in fact will correspond to deletion of subtrees of the VTS tree.

Considering the evolutionary techniques of insertion and deletion can be iterated to achieve fully incremental QE , it may be desirable to prune the tree when "backtracking" by deletion to make subsequent tree traversals less costly. Lines 23 and 61 of Algorithm 48 represent this pruning. Instead of traversing the subtree of an IQER from a now defunct test point, we remove the corresponding branch of the tree by removing that IQER as a child, and no IQERs from that subtree find themselves in relevant containers due to the lack of traversal. On the other hand, deduction that a test point is redundant, i.e. it can no longer be attributed to any constraint contained in Φ, is not free. If one is to prune the tree, lines 54 and 56 of Algorithm 48 have that we calculate the set of test points for two (simplified) Tarski formulae at any non leaf IQER traversed. The first owes to the deleted subformula, and the second owes to the whole formula after deletion. Calculation of these sets is merely to enable deduction of the test points to prune, and serves no strict purpose with respect to propagation of VTS and QE. Worse, it incurs a non trivial cost that has in theory already been expended in a previous QE operation — all these test points have already been generated, but we cannot separate out the test points to deduce which owe to the deleted formula because there is no retained correspondence between test points and atoms. The notable cost in test point generation is factorisation of potentially multivariate polynomials, via Košta's at-cs-fac on atomic formulae beneath atposl. This forms a subset of the total assumed expensive polynomial operations in VTS (leaving aside the pseudoremainders of the actual act of virtual substitution). In total, this is not

entirely in the spirit of evolutionary methods. However, the value in recalculation of these structural test point sets for the purposes of pruning must be linked to the intent of further evolutionary operations. Any defunct and unpruned test points in the tree to remain may incur undesirable cost in the future via new formulae to insert, where Algorithm 46 will substitute these defunct test points into formulae owing to inserting formulae to likely needless effect, in light of meaningful test points generated with respect to atoms that are actually present. One notes the methodology of SMT to potentially perform backtracking (decrementality), before resuming with addition of new constraints to evaluate (incrementality). Pruning of test points is in some sense a "destructive" operation with respect to the VTS tree — there exists no analogous destructive operation for decremental CAD in `QuantifierElimination` that does occurs outside of the reevaluation of cells' truth values and discarding of subtrees as a result. In other words, this is the only truly "proactive" destructive operation.

5.2 Evolutionary CAD

Strictly, incremental CAD is the only evolutionary concept *needed* to realise the poly-algorithmic QE system. The "Master CAD" idea of the poly-algorithm merely requires that the CAD can accommodate solution of similar `IQER`s, and addition of polynomials from subsequent `IQER`s is sufficient to achieve this, due to the following Lemma 77.

Lemma 77. *A sign invariant CAD for a set of polynomials owing to inequalities A, and a set of polynomials that are ECs E is also sign invariant for any subset $A' \subseteq A$ and $E' \subseteq E$ judged as polynomials from inequalities and polynomials judged as ECs respectively.*

Proof. The set arithmetic in any constituent step of projection (Algorithm 8, 9, or 10) is only ever constructive, rather than destructive (i.e. set unions, not set differences or intersections). Each canonical projection basis produced in projection (via Algorithm 5) on A and E is certainly a superset of the projection basis of the corresponding level that would be produced for A' and E'. To be precise, the canonical CAD projection sets $B_{A_i}, \forall i = 1, \ldots, n$ and $B_{E_i}, C_i \; \forall i = 1, \ldots, n-1$ (Figure 3-7) when produced from A and E are each supersets of the corresponding canonical sets that would be produced from A' and E'.

Lemma 77 enables decremental CAD immediately, considering the existing projection data structure is immediately going to suffice, and one need only worry about the CAD tree. It enables incremental CAD if one starts with A' and E' as in that lemma, and manages to extend the projection to represent A and E, and once again one need worry about the lifting. We first move to explain incremental projection & lifting to accommodate the functions used in the poly-algorithm, and evolutionary algorithms to achieve general evolutionary CAD also follow.

5.2.1 Incremental Projection

Incremental projection is the process of taking an existing `projection` data structure and modifying it to be appropriate to describe (at least, but probably not at most)

projection bases for new sets of polynomials A and E, where A and E are the usual duo of sets of polynomials for inequalities and equational constraints for a new problem.

Incremental projection in CAD is achieved via a caching approach in `QuantifierElimination`, as opposed to an approach on set arithmetic. Caching is more elegant than an approach via set arithmetic on incoming polynomials, considering the `projection` data structure stores up to three bases per canonical CAD level, B_{A_i}, $B_{E_{i-1}}$, and C_{i-1} (Figure 3-7) because of the presence of equational constraints. Figure 3-3 exemplifies the contribution of various sets to one another in an intermediate projection step with ECs, and one can imagine the combinatorics in terms of set arithmetic to begin to ensure one avoids redundant polynomial operations.

In fact, an approach using set arithmetic cannot necessarily avoid introducing redundant polynomials to the projection sets. Let B_{A_n}, $B_{E_{n-1}}$, and C_{n-1} be the canonical projection bases at the top level (level n). Let A and E be new incoming sets of polynomials at this level via incrementality, where A is a basis. Let there exist $f \in E$ such that $\prod C_{n-1} \mid f$, with $g_1 \cdots g_k$, $k \geq 1$ the factors of the quotient. Then the pivot set C_{n-1} should be extended to include $g_1, \ldots, g_k$, i.e. $C_{n-1} \leftarrow C_{n-1} \cup \{g_1, \ldots, g_k\}$. For one to be certain that all the projection bases genuinely follow from one another (to ensure the validity of future set arithmetic in incremental projection), one needs to now take (at least, and not at most) $\text{res}_{x_n}(\{g_1, \ldots, g_k\}, B_{A_n} \setminus A)$ to contribute to $B_{A_{n-1}}$, despite such resultants not being relevant to the past problem or the incoming problem as of incrementality, but would merely be present to accommodate the projection sets being canonical. Meanwhile, feeding A' and E' through all projection steps with caching introduces no redundancy in terms of polynomials, while genuinely being incremental in terms of avoiding the time complexity of computing already existing resultants and discriminants. We ensure that we receive all the polynomials necessary to construct a sign invariant CAD for A and E without redundancy in terms of operations to perform, and should pay attention to those that are genuinely new to projection bases for modification of the CAD tree in terms of lifting.

One notes that full Lazard projection without any equational constraints *can* use an approach with set arithmetic — if B_A is the past basis before a projection step, and B'_A is a new basis from incrementality, then the polynomials $PL(B'_A \setminus B_A) \cup \{\text{res}(f, g) \mid f \in B'_A \setminus B_A, g \in B_A)\}$ are the new projection polynomials for the next level. Incremental projection in `QuantifierElimination` always takes a caching approach, because we assume we may need to cater for a future case with equational constraints. The fact that the results of polynomial operations are cached also helps other CAD computations used outside the context of the current QE, i.e. if the user makes two distinct similar calls to CAD without the context of incrementality in Maple, the cached operations are retained between the calls. This is often the case for other native functions in Maple, where Maple often caches results of low level operations.

To achieve caching, the non trivial polynomial operations in Lazard projection, resultants and discriminants are cached. `QuantifierElimination` defines wrapper functions for both that associate a cache for results. The caching is destined to make incremental projection via Algorithm 54 work, because steps of projection are called on bases of polynomials that are represented normally, as is always done in basis generation in `QuantifierElimination`. The caching function for the resultant takes care to cache

symmetric calls, because $\text{res}(f, g) = -\text{res}(g, f)$, but the negative sign on the right hand side does not matter because we only care to make canonical bases in the end, where the leading coefficient is destined to be positive.

More formally, full incremental projection is described by Algorithm 50. In reality, this is very similar to full projection of all orders via Algorithm 5, where the canonical sets of polynomials A and E are fed through projection as usual, but we hope to make significant use of caching for the operations beneath individual projection steps. Otherwise, there is some additional logic to inspect presence of past and present equational constraints at individual levels, and particular attention is paid to whether usage of the same pivot can be coerced out of the present equational constraints P_E.

Proposition 78. *Algorithm 50 achieves the goal of modifying a* `projection` *object such that it contains the polynomials from all projection bases generated by the passed A and E, for A, $E \subset \mathbb{R}[x_1, \ldots, x_n]$, where A is a set of polynomials owing to inequalities, and E a set of polynomials owing to equational constraints.*

Proof. 1. The structure of the main bulk of the algorithm is essentially repeated thrice to accommodate the cases for level n, the intermediate levels where only semi restricted projection is allowable on ECs, and the last step where restricted projection is once again allowable. The only other difference is which projection level to act upon at any one time.

2. Because of the caching approach taken by incremental projection, much of the structure of Algorithm 50 is the same as that of standard projection (Algorithm 5). The idea is to perform projection on the top level sets in whatever context those sets arise in incrementality, and any of the expensive operations such as resultants and discriminants are cached such that the projection genuinely is incremental. One performs a "set difference" on the *bases* acquired in this process returned in the `Array` incBases are entirely of new polynomials to the CAD. These set differences are slightly frustrated by the fact that the sets stored in the `projection` object at any one level are slightly inhomogeneous (the sets of equational constraints may not exist when ECs were not used at that level). Hence practically it is more convenient to iterate over the level of the `projection` object to remove any polynomials already existing to make the sets disjoint. Practically, any line reading "incBases$[i] \leftarrow \ldots \setminus D$ where D is the set of all polynomials in P at level $n - i + 1$" is achieved in this way. Because the polynomials all receive full factorisation to create bases, we rely on the canonicality of the representation of polynomials to ensure polynomials to be stored in incBases are coprime to those from projection. One notes the caches of polynomial operations are cleared whenever it can be deduced that the polynomials to come through projection in any circumstance do not largely coincide with the cached polynomials. The only circumstance where this actually comes to fruition is in the poly-algorithm whenever a new CAD is generated (Code Fragment 42).

3. The algorithm does not attempt to extend the projection bases to represent strict supersets of what was input at the top level in terms of A and E from the past

Algorithm 50 Incremental Projection algorithm via Caching

Input: P, a `projection` object for an existing CAD, A a set of all polynomials associated with inequalities from input as of incrementality, E a set of all ECs from input as of incrementality, vars, n the total number of CAD variables as of incrementality, new_m the number of new quantified variables, new_n the number of new free variables, UseGroebner a keyword option as boolean flag dictating if Gröbner bases should be used, PropagateECs a keyword option as boolean flag dictating if ECs should be propagated, and UseEquations a keyword option as symbol `'none'`, `'single'` or `'multiple'` defining the capacity for equational constraints usage in restricted projection operations

Output: incBases, an `Array` of canonical irreducible polynomial bases of each level of new polynomials from incrementality coprime to those from the existing `projection` data structure P, and newPivots, an `Array` of boolean values representing if a new pivot was used at the canonical CAD levels 2 through $n - 1$

1: **procedure** PROJECTIONINCREMENTAL(P, A, E, vars, n, new_m, new_n, UseGroebner, PropagateECs, UseEquations)
2: Initialise incBases as an empty `Array` with n elements
3: Initialise newPivots as an `Array` of $n - 1$ elements, all *false*
4: (P_A, P_E) $\leftarrow A, E$
5: **if** UseEquations = `'none'` **then**
6: $P_A \leftarrow P_A \cup P_E$
7: $P_E \leftarrow \emptyset$
8: **end if**
9: **for** i **to** new_n **do**
10: Append $\emptyset$ to $P \mapsto$ inequalities
11: Append 0 to $P \mapsto$ equations
12: Append 0 to $P \mapsto$ pivotSets
13: **end for**
14: **for** i **to** new_m **do**
15: Prepend $\emptyset$ to $P \mapsto$ inequalities ▷ As opposed to *appending* above
16: Prepend 0 to $P \mapsto$ equations
17: Prepend 0 to $P \mapsto$ pivotSets
18: **end for**
19: **if** UseGroebner **and** $|P_E| > 0$ **then**
20: $P_E \leftarrow$ equationalConstraintsToGroebner(P_E, vars)
21: **if** $P_E = \{p\}$ for some polynomial p, **and** $\deg(p) = 0$ **then** ▷ Likely $P_E = \{1\}$, but equally $\{c\}$, any $c \in \mathbb{R} \setminus \{0\}$
22: incBases $\leftarrow$ an `Array` of n empty sets
23: **return** incBases, newPivots▷ To exit similarly to line 10 of Algorithm 5
24: **end if**
25: **end if**

Algorithm 50 Incremental Projection, Part 2

26: **if** $n > 1$ **then**
27: $x \leftarrow \text{vars}[-1]$ ▷ $x = x_n$
28: **if** There exists p in P_E such that $\deg_x(p) > 0$ **then**
29: **if** $P \mapsto \text{equations}[-1]$ is a set, i.e. a pivot set was previously used at level n **then**
30: $(\ P_E, \text{cont}_E\ ) \leftarrow \{f \mid f \in P_E, \deg_x(f) > 0\}$, $\{f \mid f \in P_E, \deg_x(f) = 0\}$
31: **if** There exists p in P_E such that $p \mid \prod_{f \in C_{n-1}} f$ **then** ▷ $C_{n-1} = P \mapsto \text{pivotSets}[-1]$
32: Factor p as $\text{piv}_c \cdot q_1 \cdots q_k$ where $\deg_x(\text{piv}_c) = 0$, $\deg_x(q_j) > 0$, $j = 1, \ldots, k$, the q_j pairwise coprime
33: $P_E \leftarrow P_E \setminus \{p\}$
34: $(\ B_A, \text{cont}_A, B_E\ ) \leftarrow$ CADMakeBasisWithEqns(P_A, P_E, $\{q_1, \ldots, q_k\}$, x)
35: $(\ P_A, P_E\ ) \leftarrow$ lazardProjectionRestricted(B_A, P_E, $\{q_1, \ldots, q_k\}$, piv_c, cont_A, cont_E, x, PropagateECs)
36: $\text{incBases}[-1] \leftarrow (\ B_A \cup B_E\ ) \setminus D$ where D is the set of all polynomials in P at level n
37: $P \mapsto \text{inequalities}[-1] \leftarrow P \mapsto \text{inequalities}[-1] \cup \text{incBases}[-1]$
38: **else** ▷ Can't use ECs in restricted projection — would fail to track at least one of the pivot sets at this level
39: $(\ B_A, \text{cont}_A, B_E\ ) \leftarrow$ CADMakeBasisWithEqns(P_A, P_E, $\emptyset$, x)
40: $(\ P_A, P_E\ ) \leftarrow$ sets obtained by Lazard projection on B_A and B_E, with P_E all propagated equational constraints via resultant rule from $\text{res}_x(B_E, B_E)$ if PropagateECs = *true*, else $\text{res}_x(B_E, B_E) \subset P_A$
41: $(\ P_A, P_E\ ) \leftarrow P_A \cup \text{cont}_A, P_E \cup \text{cont}_E$
42: $\text{incBases}[-1] \leftarrow B_A \cup B_E \setminus D$ where D is the set of all polynomials in P at level n
43: $P \mapsto \text{inequalities}[-1] \leftarrow P \mapsto \text{inequalities}[-1] \cup \text{incBases}[-1]$
44: **end if**
45: **else** ▷ First time observing equational constraints at level n
46: $(\ B_P, P_E, \text{cont}_E, \text{piv}_c\ ) \leftarrow$ choosePivotSet(P_E, x)
47: $(\ B_A, \text{cont}_A, B_E\ ) \leftarrow$ CADMakeBasisWithEqns(P_A, P_E, B_P, x)
48: $(\ P_A, P_E\ ) \leftarrow$ lazardProjectionRestricted(B_A, P_E, B_P, piv_c, cont_A, cont_E, x, PropagateECs)
49: $\text{incBases}[-1] \leftarrow (\ B_P \cup B_E \cup B_A\ ) \setminus D$ where D is the set of all polynomials in P at level n
50: $(\ P \mapsto \text{pivotSets}[-1], P \mapsto \text{equations}[-1]\ ) \leftarrow B_P, B_E$
51: $P \mapsto \text{inequalities}[-1] \leftarrow P \mapsto \text{inequalities}[-1] \cup B_A$
52: $\text{newPivots}[-1] \leftarrow$ *true*
53: **end if**

Algorithm 50 Incremental Projection, Part 3

54: **else** ▷ No equational constraints to use at all in x_n
55: (B_A, cont_A) ← CADMakeBasis(P_A, x)
56: $P_A \leftarrow$ lazardProjection(B_A, cont_A, x)
57: incBases[-1] ← $B_A \setminus D$ where D is the set of all polynomials in P at level n
58: $P \mapsto$ inequalities[-1] ← $P \mapsto$ inequalities[-1] $\cup B_A$
59: **end if**
60: **end if**
61: **if** UseEquations = `'single'` **then**
62: $P_A \leftarrow P_A \cup P_E$
63: $P_E \leftarrow \emptyset$
64: **end if**
65: **for** i **from** 2 **to** $n-2$ **do**
66: Use P_A and P_E to extend projection at level $n-i+1$ in much the same way as the case for x_n above, attempting to coerce usage of existing pivot sets where they exist. However, use semi restricted projection in place of restricted projection, and $x_{n-i+1} = \text{vars}[-i]$ in place of x_n
67: incBases[$-i$] is the fully factored basis of all new level $n-i+1$ polynomials coprime to those that were already in projection at level $n-i+1$
68: newPivots[$-i$] is the boolean value corresponding to if a new pivot set was introduced at level $n-i+1$
69: **end for**
70: **if** $n > 2$ **then**
71: Use P_A and P_E in projection in much the same way as the case for x_n, attempting to coerce usage of an existing pivot set if one exists. Usage of restricted projection with ECs is once again allowable, but we use $x_2 = \text{vars}[2]$ in place of x_n
72: incBases[2] is the fully factored basis of all new level 2 polynomials coprime to those that were already in projection to level 2
73: newPivots[2] is the boolean value corresponding to if a new pivot set was introduced at level 2
74: **else**
75: $P_A \leftarrow P_A \cup P_E$
76: **end if**
77: (B_A, __) ← CADMakeBasis(P_A, vars[1]) ▷ vars[1] = x_1
78: incBases[1] ← $B_A \setminus P \mapsto$ inequalities[1] ▷ $P \mapsto$ inequalities[1] the only set that exists at level 1 in a `projection` object
79: $P \mapsto$ inequalities[1] ← $P \mapsto$ inequalities[1] $\cup$ incBases[1]
80: **return** incBases, newPivots
81: **end procedure**

computation to the new computation. Instead, one can view projectionIncremental as adding all those projection polynomials owing to the new A and E, and deducing all the new ones per level such that one knows what polynomials to extend the lifting with to accommodate the new A and E.

4. Control of equational constraints is similar to the case for Algorithm 5, via UseEquations, PropagateECs and UseGroebner, corresponding to the values of the identically named keyword options. These options need not be the same as what was used throughout the last run of projection.

5. When equational constraints were previously used at a particular level, we pay attention to the current set of equational constraints P_E to check if any element p of P_E divides the previously used pivot (and is hence contained within the existing pivot set at that level after factorisation). This can be seen on line 31, with similar checks occurring at levels 2 through $n-1$. If an element does divide the existing pivot set, we coerce usage of that element as pivot in (semi-)restricted projection to try to reuse as many of the cached resultants as possible. The set of factors of p, and the content of p, piv_c take the role as pivot in (semi-)restricted projection. This of course covers the case where p is equal to the existing pivot. The existing pivot set stored in the `projection` object is sufficient to remain intact because of the division, and the state of play with respect to existing cells and curtains remains the same. This is in contrast to usage of choosePivotSet (Algorithm 22), which may choose a different pivot (ostensibly better by some metric) but producing new distinct polynomials which we needn't produce, and more importantly inducing difficulties with canonicality of the projection object in terms of stored pivot sets (note 9 in this proof). Note that as P_E isn't a basis, it is more correct to check if any of its polynomials divide the pivot rather than checking if $|P_E \cap C_{n-i}| > 0$. However, P_E can and should contain as much information about factorisations as possible (polynomials produced by the resultant rule via propagation of equational constraints should be stored in factored form), and one can clearly produce $\prod_{f \in C_{n-i}} f$ in factored form without expansion to retain as much factorisation information in these checks for division as possible. One regret is that pivots are stored in factored form without their content in $P \mapsto \text{pivotSets}[i]$, so if a pivot had a content of non trivial degree when first factored we may fail to realise that a new EC divides such an existing pivot, if the new EC also attributes a non trivial content. Ideally this would be improved such that we can be sure the effects of ECs remain in CAD when we truly do use the same equational constraints.

6. New polynomials generated at any one level are added to the appropriate sets stored in the `projection` object once the appropriate set difference has been deduced, such that the `projection` object reflects the bases after incrementality.

7. Incremental projection can accommodate polynomials in further variables than were used in projection before. The variable ordering in terms of all variables used previously must remain fixed such that the existing projection bases before incrementality remain canonical. That is, we cannot insert a new variable

in between existing levels. One can however prepend new quantified variables, and append free variables, although appending of free variables implies that the CAD tree must be relifted. The polynomials then feed through accordingly for the variable ordering passed which must include the new variables prepended & appended to the past variable ordering in the way described. In practice, only poly-algorithmic QE induces the case where new quantifiers are defined, by repurposing a CAD from one `IQER` for another `IQER` of a lower level (which attributes more quantified variables).

8. Gröbner bases are taken on the incoming set of equational constraints E if requested via `'UseGroebner'`. The methodology and reaction to output is much the same as for Algorithm 5. In the case of pure CAD incrementality by clauses, P_E is still the set of all equational constraints, so addition of an equational constraint to a formula that is inconsistent with previous ECs will reset the projection to something trivial at this point, and the output incBases is an `Array` of n empty bases. This ensures as little new non trivial geometry as possible is built as a result — in the case of a relift, the CAD will lift to one cell representing $\mathbb{R}^n$, certainly holding the truth value *false*. Otherwise, incremental lifting reduces to tree traversal on the existing CAD tree, where cells with determinate truth values can only hold *false*. Unevaluated cells or those with indeterminate truth values added to cad for further stack construction only lift $\mathbb{R}$ per level due to the lack of polynomials in any projection basis until a child cell deduces the determinate truth value *false*. Usage of Gröbner bases for preprocessing in incremental CAD should ideally remain consistent with what was used as the option for preprocessing in the last pass of CAD (i.e. the keyword option `'UseGroebner'` should remain consistent) to reconcile with the caching approach of incremental projection.

9. We cannot use a pivot set in (semi-)restricted projection at some level where a pivot set already exists, and the pivot set to use would not be a subset of the existing pivot set (line 38). It is unclear how one would store the new pivot set in light of the existing pivot set in the `projection` object, which is vital in terms of identification of curtains — via the object methods provided in this work, identification of curtains always falls to examination of the set $P \mapsto \text{pivotSets}[i]$ for i the canonical CAD level. In the case where ECs exist, but we cannot use them in restricted projection due to an existing pivot set, we perform standard Lazard projection, but propagate ECs via the resultant rule on the resultants between pairs of elements in P_E, such that we may attempt to use propagated ECs in restricted projection later when appropriate.

10. However, if a pivot set did not exist at some level, and new ECs exist at such a level, then we can define a new pivot set (line 45, and similarly at other levels). newPivots is an `Array` tracking which canonical CAD levels had a new pivot set introduced where one did not exist previously. This allows us to restrict checking for curtains in tree traversal for cells at as few levels as possible later (e.g. Algorithm 52), because we know existing cells have already undergone checking for curtains on existing pivot sets, so needn't be checked on such pivots

again. In total, the pivots stored in $P \mapsto$ pivotSets can represent pivots from distinct overlapping runs of projection.

5.2.2 Incremental Lifting

Next are suitable methods to describe "incremental lifting" - that is, adapting a previously computed CAD tree to reflect the additional projection basis elements contributed via the new formula from incrementality, which are deduced and returned from Algorithm 50. As such, this should follow incremental projection as a step in incremental CAD. It is important that incremental projection produces an object corresponding to projection polynomials completely new to projection per level. These two algorithms are sufficient to have all the constituent parts to realise the CAD repurposing in poly-algorithmic QE via Algorithm 44, belatedly concluded here. Additionally this algorithm is used in the context of curtain decomposition (e.g. Algorithm 31). Algorithm 51 is a key driver in incremental lifting in any context.

Algorithm 51 is in some sense an amalgamation of Algorithms 18 and 14. Those algorithms act in the non incremental sense, whereas this acts in the incremental sense that it acts on a cell c that must have child cells. From B_{new}, a basis of univariate lifting polynomials obtained from c and a basis of projection polynomials of appropriate level new to this iteration of CAD incrementality, we isolate the roots of such lifting polynomials via usual technology. These roots should be distinct from roots used previously to construct the cells below c, and certainly will be if all the lifting polynomials used were canonical. However, they may not have disjoint isolating intervals to those used in the stack below this cell, which can be deduced from the bounds of the child cells. Hence we may need to perform root refinement to make such intervals disjoint. From genuinely new root descriptions, we can *modify* bounds of existing child cells, and create new unevaluated cells to "merge" in the appropriate geometry such that the stack below c genuinely reflects the current projection sets as of incrementality. In other words, incrementalCADMerge merges new cells owing to the new lifting polynomials from B_{new} below the cell c. Further details are given in proof of proposition of the algorithm.

Proposition 79. *Let c be a cell with child cells $c \mapsto$* children*. B_{New} is a univariate polynomial basis received via Lazard evaluation of a multivariate set B of polynomials of level $n - c \mapsto$* level *owing from the Array* incBases *returned from Algorithm 50. T maps polynomials in B_{New} to that set B.*

Algorithm 51 successfully merges in new cells amongst the existing child cells of c according to real roots from B_{New}, which may modify existing child cells. New cells are merged into $c \mapsto$ children*, which remains ordered with respect to the local indices of such cells.*

Proof. 1. incrementalCADMerge is an amalgamation of the purposes of both isolateRootsBasis (Algorithm 18) and CCHILD (Algorithm 14), but in an evolutionary context. c is assumed to have child cells, which implies it evaluated with an indeterminate truth value (if truth values are relevant, i.e. QE by Partial CAD,

Algorithm 51 Incremental CAD Merge Algorithm

Input: c, a `CADCell` for which child cells should be modified & added to via new roots from B_{New}, B_{New}, a single basis of univariate polynomials in x received from Lazard evaluation of *new* projection polynomials, T a table mapping polynomials in B_{New} to the multivariate polynomials they came from before Lazard evaluation, x the canonical local variable $x_{n-c\mapsto\text{level}}$, "lvl" is the level of all child cells of c. "open" a boolean flag where its truth indicates we do not build new sections, "constraints" and "bounds" `Array`s created from parsing of lifting constraints (when passed), cad a container to store *new* unevaluated cells (when passed),

Output: No meaningful return value, but implicitly modifies the CAD tree via possible modification of child cells of c and addition of new ones.

1: **procedure** INCREMENTALCADMERGE(c, B_{New}, T, x, lvl, open, constraints, bounds, cad)
2: **if** open **then**
3: newInc $\leftarrow 1$
4: **else**
5: newInc $\leftarrow 2$
6: **end if**
7: **for** b **in** B_{New} **do**
8: isolated $\leftarrow$ isolateRootsOf(b, x, constraints, bounds)
9: (i, b') $\leftarrow 1, b$
10: **for** [lb, ub] **in** isolated **do**
11: rtf $\leftarrow$ RootOf(b', lb..ub)
12: nc $\leftarrow |c \mapsto \text{children}|$
13: **repeat**
14: **while** $i \leq \text{nc} - 1$ **and** $c \mapsto \text{children}[i] \mapsto \text{local\_index}$ **mod** $2 = 0$ **or** $c \mapsto \text{children}[i] \mapsto \text{local\_index}$ **mod** $2 = 1$ **and** $\text{lb} > c \mapsto \text{children}[i+1] \mapsto \text{lowerbound}$ **or** $\text{lb} > c \mapsto \text{children}[i+2] \mapsto \text{lowerbound}$ **do**
15: i++
16: **end while**
17: **if** $i \leq \text{nc} - 1$ **then**
18: $\text{cell}_{\text{left}} \leftarrow c \mapsto \text{children}[i]$
19: **if** $c \mapsto \text{children}[i+1] \mapsto \text{local\_index}$ **mod** $2 = 1$ **then** ▷ Is the next cell along a local sector?
20: $\text{cell}_{\text{right}} \leftarrow c \mapsto \text{children}[i+1]$
21: **else**
22: $\text{cell}_{\text{right}} \leftarrow c \mapsto \text{children}[i+2]$
23: $\text{cell}_{\text{central}} \leftarrow c \mapsto \text{children}[i+1]$ ▷ The local section in between the local sectors
24: **end if**
25: pastLB $\leftarrow$ the upper bound of the interval from $\text{cell}_{\text{left}} \mapsto$ upperbound
26: pastUB $\leftarrow$ the lower bound of the interval from $\text{cell}_{\text{right}} \mapsto$ lowerbound

although incrementalCADMerge can be called in the context of full CAD in curtain recovery — Algorithm 31). Existing root descriptions need to be gleaned from existing child cells below c, as they exist as the lower and upper bounds of sectors below self. Those lower and upper bounds are rational numbers or RootOfs representing irrational numbers with an isolating interval, i.e. real algebraic numbers. This is why the iteration of line 14 pays attention to whether the cell in question is a sector or not, as only sectors store lower and upper bounds. CCHILD outsources creation of a container of sorted disjoint isolating intervals to isolateRootsBasis before easily creating the cells from those root descriptions. Here, root descriptions already exist via existing cells, so we use the lower and upper bounds of those cells as the equivalent of the containers forming root descriptions gleaned from isolateRootsBasis. We know that they are sorted, i.e. the child cells line up from left to right along the real line with respect to x, and hence the root descriptions to examine are also sorted in the same way as they would be as of output of isolateRootsBasis and hence in CCHILD.

2. B_{New} is a set of lifting polynomials inherited from Lazard evaluation on a subset of projection polynomials — in particular new ones gleaned from Algorithm 50. The projection polynomials from B are distinct from those used in the past when lifting around c (they result from set differences of new and existing polynomials from projection in Algorithm 50, and we note Remark 34), but the results of Lazard evaluation may not be distinct to those used before, but new generated root descriptions that coincide with an existing used root description are dealt with by refineIsolatingIntervals, which attempts to deduce if the root descriptions are the same. Hence we only merge in new geometry.

3. Replacement of the upper bound of a cell (Line 88) to accommodate a new root from a new polynomial is one of the key parts of this algorithm. replaceUpperBound is a `CADCell` method that:

   - replaces the static upper bound "upperbound" of a cell with the real algebraic number rtf,
   - replaces the upper bound from the local cell description of the cell with the real algebraic function mRtf,
   - unevaluates the cell by resetting its local sample point (which may not be valid any more due to the smaller local interval formed by the new bounds in light of the full sample point of its parent cell),
   - protects its truth value by setting it to FAIL (if it wasn't protected already),
   - and discards the existing CAD subtree below the cell.

   Algorithm 51 incrementalCADMerge is usually called from the context of some sort of tree traversal, such as Algorithm 52. Hence it is the job of the calling function of incrementalCADMerge to identify these newly unevaluated cells and treat them accordingly, by reevaluating their local sample point, Tarski formula,

and truth value. If the truth value is still indeterminate (FAIL) after this evaluation then the cell should be readded to the container of unevaluated cells for later stack construction, else it is added to a container of leaf cells.

4. In this way, cells undergoing bound replacement can be identified as Lazard curtains later at the regular time of stack construction by CCHILD in the regulation lifting that occurs in evolutionary CAD, much like the non evolutionary case. The same is true of new cells formed by incrementalCADMerge — despite the flat presentation of polynomials to incrementalCADMerge in incBases that contains no information about whether any of these polynomials are pivots, the cells are checked for curtains on whatever pivot sets are used at the point of stack construction later (which is certainly useful in the context of Partial CAD). Delaying the attempt to identify a newly unevaluated cell as a Lazard curtain to the calling of CCHILD (Algorithm 14) is appropriate to keep the overall CAD lifting process homogeneous, and fits in with the strategies for avoidance of & recovery from Lazard curtains (Section 3.7.2), which in the evolutionary case still assumes regulation lifting as far as possible before attempting classification and then recovery from curtains. The only difference with the evolutionary case is that regulation lifting may be preceded by some sort of tree traversal such as Algorithm 52.

5. Replacement of lower/upper bounds is not to be confused or conflated with refinement of lower/upper bounds (e.g. Line 30), where we examine the isolating interval for a new root isolation and attempt to make it disjoint from that for an existing root isolation used as a (static) upper and lower bound for $\text{cell}_{\text{left}}$ and $\text{cell}_{\text{right}}$ respectively. The existing interval about the root can only become smaller as a result of usage of refineIsolatingIntervals (see Section 3.4.1), hence the sample points for $\text{cell}_{\text{left}}$, $\text{cell}_{\text{right}}$, and $\text{cell}_{\text{central}}$ remain valid and we needn't unevaluate these cells. As per discussion in Section 3.4.1, we retain any information about refinement of isolating intervals about real algebraic numbers, mostly here by overwriting existing properties of various cells.

6. incrementalCADMerge acts "right-on" — that is, new cells are always inserted "to the right of" an existing cell, and one notes that we replace (at most) upper bounds of existing cells, as we implicitly move across the real line from left to right, attempting to merge in root descriptions and new cells as we go. If we create the new right-most cell on Line 79, we need to take into account the right-most bound owing to local lifting constraints, where they were passed.

7. When iterating amongst the child cells, we intend to find two neighbouring local sectors, which share cell bounds. Sectors have odd local cell indices. This is the loop on line 14, but we do not assume the order of child cells of c is "sector, sector, sector..." or "sector, section, sector...". incrementalCADMerge offers support for merging of non open geometry into an open CAD, to accommodate for example the freedom usually offered by full CAD (CylindricalAlgebraicDecompose). However, merging of non open geometry into an open CAD is *not* correct in the case for QE, because we may miss (local) sections owing to projection polynomials

from a past computation (i.e. those not in incBases), and incrementalCADMerge does not identify that they should now be created — doing so would require some bespoke examination of the "missing" local sections in terms of surrounding local sectors, and deduction of the information to use in order to create such sections.

Algorithm 52 is another recursive tree traversal algorithm, traversing from the root cell down. Its main purpose is to repurpose the CAD tree to evaluate truth values of a different real Tarski formula, Ψ. Because Ψ may have contributed new distinct projection polynomials, we also merge in new geometry at appropriate cells arising from real roots of lifting polynomials attributed to said new projection polynomials in incBases. Its methodology is not dissimilar from the VTS tree traversal algorithms in that we may truncate the CAD tree at a cell that now has a determinate truth value, or identify a `CADCell` as no longer being a leaf cell (and so amenable to further stack construction) if its truth value is not determinate via the new formula to evaluate. Additionally, the formula's evaluation happens largely via recursion, such that each `CADCell` only evaluates a formula in terms of its canonical variable. We must pay attention to where new pivots have been introduced in projection, in case we traverse to a cell that should now be identified as a curtain.

Proposition 80. *Algorithm 52 traverseCADTreeModify repurposes the CAD subtree rooted at* self *such that its `CADCells` evaluate truth values for Ψ, and merges in new geometry owing to new projection polynomials owing to Ψ by* incBases *generated by Algorithm 50 at every traversed cell via Algorithm 51 incrementalCADmerge, such that calling traverseCADTreeModify at the root cell lets the CAD subtree describe a sign invariant CAD for Ψ after full tree traversal.*

Proof.

1. traverseCADTreeModify is a "repurposing" of the CAD tree. In particular the tarski_formulae and truth_values of each cell in the tree should now be *correct* in light of the top level formula to repurpose with after full traversal of the tree via this algorithm. As a result of full tree traversal, the tree may be sufficient to describe the quantifier free equivalent of the quantified formula at the top level, and if not regulation lifting must be continued outside of this function. In order to do regulation lifting, the top level requires a new collection of unevaluated cells. The two canonical mutable containers for `CADCell`s, "cad" and "leaves" appear as arguments to traversal. These are redefined as empty at the top level, and usage of this algorithm to traverse the CAD tree repopulates these containers with appropriate unevaluated and leaf cells respectively. Because this algorithm represents a "repurposing", no information about the existing formula held by any cell can or should be reused, i.e. their formulae are overwritten.

2. Not unlike Algorithm 46, Ψ is evaluated at the full sample point for self via traversal through the tree. This happens via recursion, i.e. for a cell of general level, the received Ψ is actually the evaluation of Ψ at the full sample point of the cell above, and one only needs to evaluate Ψ at the local sample point of self rather than its full sample point (which would be more costly). This covers the reevaluation of cells' formulae, and usage of incrementalCADMerge covers

merging in of new geometry owing to new projection polynomials inherited from the top level Ψ via Lazard evaluation of the appropriate set from incBases.

3. Line 12 refers to the case where we traverse to an unevaluated cell. Such an unevaluated cell may:

   - have remained unevaluated from previous CAD computation (i.e. it was left unevaluated by Partial CAD in a previous iteration of QE)
   - have been unevaluated by usage of incrementalCADMerge, where the upper bound of a cell is replaced by a new real root (Line 88, Algorithm 51)
   - have been created by usage of incrementalCADMerge (again, due to presence of new incoming real roots)

   In this case we evaluate the cell (i.e. deduce and set its sample point, (real) Tarski formula, and truth value, in that order) via usage of evalAndSetTruthValue after ensuring the truth value really is unset, to force evalAndSetTruthValue to reevaluate these properties. Having evaluated the cell, it either holds a genuine boolean truth value, i.e. it is a leaf cell and evalAndSetTruthValue has added it to the container leaves, else it is amenable to further stack construction, and hence is added to the container cad.

4. It is important to call incrementalCADMerge before traversal through the subtree below (i.e. recursion on traverseCADTreeModify amongst self's child cells. Usage of incrementalCADMerge may modify a child cell c below such that its upper bound changes, and hence c's sample point may need to be reset as a result. If we were to recurse first, then the sample points of such a child cell c may be ostensibly wrong in light of future usage of incrementalCADMerge that could change its bounds, and hence invalidate its entire CAD subtree, which we are erroneously repurposing as if the geometry is certainly valid.

5. The loops on lines 5 and 50 are what is referred to as *protection of truth values*. This algorithm calls Algorithm 15 PRPTV having traversed any cell which came to have an indeterminate truth value and child cells. This is to ensure the container of unevaluated cells cad has only the necessary cells for any further stack construction as of termination of tree traversal. However, said child cells may hold a meaningful truth value from a previous computation (potentially propagated from cells in the subtree of any of those cells). In order to ensure that past information does not result in propagation of erroneous truth values in light of the new formula to repurpose the CAD tree with, we temporarily overwrite the truth values of child cells with FAIL, which holds the meaning "truth value deduced to be indeterminate". This is opposed to unsetting the truth values, which would encourage PRPTV to set them potentially in terms of erroneous data. We only protect the truth values of all but the first child cell, because we traverse through the first child cell first, i.e. move to deduce its new truth value. Later, when we traverse a cell which had a protected truth value, we properly deduce its truth value as of the new incoming formula to repurpose the cell with. Figure 5-2

demonstrates how a CAD tree may look in terms of truth values as of traversal to the first level n cell. Because we call PRPTV at every cell that still has child cells after traversal, and the recursion in terms of tree traversal happens root down, despite protection of truth values, the truth values of any meaningful leaf cell after traversal *does* propagate towards the root. The purpose of protection of truth values is to prevent incorrect truth values making their way to the root from parts of the tree that have not been processed yet.

6. Lines 4 and 43 are to accommodate truth values of child cells below changing as a result of a different formula being used. The root cell must have children cells to have even commenced building a non trivial CAD, and at line 43 we know child cells exist.

7. Line 57 accommodates the case for a cell which previously held a boolean truth value (in other words, it was previously a proper leaf cell), however upon repurposing now has an indeterminate truth value. Hence it is unevaluated, and we can build a stack over it, hence it is added to cad.

8. Line 26 reflects the need to check for a Lazard curtain on the traversed cell when a *new* pivot was introduced by incremental projection (Algorithm 50) at the projection level corresponding to this cell. This is predicated by an element of the `Array` "newPivots" produced by projectionIncremental such that we only do this when a new pivot was introduced. When the cell is a curtain, we identify it as a curtain as usual by addition to problemCells, to reconcile with the aim to avoid and/or eventually recover from lifting failures. We also unevaluate it in case it is a level $n-1$ curtain to be processed by Algorithms 32 and 34, to match the fact that curtain cells found in regulation lifting are similarly unevaluated sans retaining a local sample point.

Having delineated all the algorithms required for poly-algorithmic QE, we can belatedly prove the validity of Algorithm 44.

Proposition 81. *Given an `IQER` I and `CADData` C, Algorithm 44 successfully produces the quantifier free equivalent of I by repurposing the CAD.*

Proof. In this case, projectionIncremental actually receives polynomial sets decomposed from the formula for I, and because of the caching approach, the projection bases for I end up "coexisting" amongst polynomials from previous projection bases. Due to the "poly-share criteria", we assume there is a significant intersection between the polynomials generated from projection on I and the existing projection bases at every level, but this is an efficiency issue unrelated to the validity. Incremental projection outputs bases of polynomials known to be new to each level, and disjoint from previously used projection polynomials in lifting. The rest of the algorithm follows from Algorithm 52 and Algorithm 51, with traverseCADTreeModify reevaluating the truth values of cells in the lifted CAD tree for the formula held by I, and instigating merging in of new geometry via incrementalCADMerge. We receive containers of unevaluated

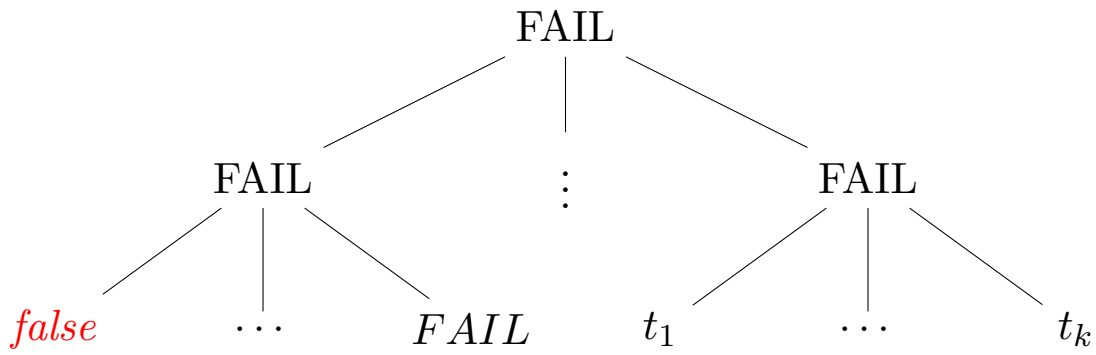


Figure 5-2: A CAD tree visualising only the truth values of cells, as of first traversal via Algorithm 52 to a level $n = 2$ cell c highlighted in red. The tree is assumed to be for a fully existentially quantified problem. traverseCADTreeModify, and in general any CAD tree traversal function which involves reevaluation of truth values overwrites truth values of nearby unprocessed cells in such a way that we cannot propagate erroneous truth values from those cells in light of the fact they do not evaluate the formula that we wish them to (yet). The truth values $t_1, \ldots, t_k$ can be any boolean value or FAIL owing to a past evaluation without being able to interfere with any other truth values if one is to attempt to call PRPTV on the parent of c.

cells and leaf cells, and any further lifting and curtain recovery resumes in the case for normal Partial CAD, such that we are eventually able to deduce the quantifier free formula for I from the container of leaf cells.

Next is an algorithm somewhat analogous to Algorithm 46, but for CAD, to modify the CAD tree for insertion. Again, this is a recursive tree traversal algorithm acting root down. Much of its structure and logic is similar to Algorithm 52, but instead of having the CAD tree evaluate a potentially entirely distinct formula, we perform insertion of a new formula by atomic position. Hence structural formulae enable us to retain the information about the evaluation of formulae on cells. Much as the same case for VTS, the formula is evaluated by recursion, such that each cell evaluates a formula in terms of just its canonical variable via its sample point.

Proposition 82. *Algorithm 53 inserts the formula Ψ' into the structural formula for each* `CADCell` *in the subtree rooted at c, where Ψ' is the evaluation of Ψ at the full sample point of c. In doing so, every* `CADCell` *holds truth values for Ψ. Additionally, we merge in new geometry owing to new projection polynomials owing to Ψ by* incBases *generated by Algorithm 50 at every traversed cell via Algorithm 51 incrementalCADmerge.*

Proof. 1. Much of the structure of the algorithm is the same as Algorithm 52, except for what is done with the formulae associated to each cell. Rather than replacement of these formulae by another potentially different formula (repurposing), we use that we merely need to insert a new formula into the structural representation of the formula in the cases where we know that the sample point for this cell is valid, i.e. when we reach cells that have remained evaluated, hence

not unevaluated by usage of incrementalCADMerge. By simplifying this unstructural formula, we receive the cell's tarski_formula, hence able to rededuce its truth value on the top level formula after insertion.

2. The algorithm also shares similarities with its counterpart for VTS, Algorithm 46. Because of structural formulae, we only need to evaluate the new formula to insert at the sample point for self, rather than evaluation of a whole formula, and this evaluation happens via recursion such is the case for Algorithm 52.

3. Due to similarities of the algorithms, most remarks on Algorithm 52 apply similarly to Algorithm 52. This includes protection & propagation of truth values (lines 5 and 5, example viz: Figure 5-2) which occurs for similar reasons as in Algorithm 52. Changing formulae in cells of the tree mean that their truth values are uncertain until the whole subtree beneath them has been fully traversed.

4. Other almost identical facets as in the case for traverseCADTreeModify are treatment of unevaluated cells, checking for Lazard curtains on existing cells where there is now a new pivot in projection at the relevant level.

5. No insertion features on the (real) Tarski formula held by the root cell — `QuantifierElimination` does this before tree traversal ever happens, because it is required in order to rededuce equational constraints before incremental projection.

Via incremental projection and incremental lifting to achieve insertion, we can realise the counterpart of incremental poly-algorithmic QE via insertion of formulae for Partial CAD, Algorithm 54.

Proposition 83. *Usage of Algorithm 54 CADIncremental achieves the goal of incremental QE by Partial CAD, enabling insertion of a new formula ψ into Φ at atomic position α with operation* oper *for a past quantified QE problem $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi$ donating* `CADData` data *from that elimination, such that we can produce QE output for the problem after insertion.*

Proof. We insert the new formula into the Tarski formula held by the root cell before tree traversal and incremental projection, such that we can decompose the new top level formula after insertion to rededuce equational constraints for passing to incremental projection (doing so without usage of getPolySets is difficult, because we allow for insertion at arbitrary atomic position). In this case, incremental projection genuinely does feed supersets of past polynomials fed through projection to extend upon projection. In this way, if there was a polynomial that was previously an equational constraint, if it is now identified as not an equational constraint due to insertion with atomic position, usage of Algorithm 50 populates the projection bases with appropriate polynomials in light of the fact this equational constraint is missing, because the logic is on the current set of equational constraints, and so if restricted projection cannot be done with the sets currently held, we receive the necessary polynomials without restricted projection.

Similarly to the case for proof of validity of Algorithm 47, usage of projectionIncremental into traversal of the entire CAD tree via traverseCADTreeInsert is sufficient to make the CAD tree *correct* in light of the top level formula after insertion, but may not necessarily be sufficient to describe quantifier free output. If it isn't, we commence with standard regulation lifting into lifting recovery with respect to the held projection bases.

CADIncremental offers the option to completely relift the CAD. This is not strictly in the spirit of incrementality, but the option is offered to discard the existing tree, perhaps because it is too fine due to successive usage of incrementality. In this case, the `projection` object contains projection bases sufficient to build a sign invariant CAD for the formula, so a newly lifted tree accommodating the formula after insertion follows. If the formula to insert contained free variables ($k > 0$), incremental projection is canonical, but the past lifted CAD fails to be, so we must relift (see discussion following Algorithm 41).

5.2.3 Decremental CAD

Lemma 77 implies that modification of the tree in terms of addition or removal of cells (i.e. CAD tree nodes) is unnecessary when one requires a sign invariant CAD for pairwise subsets of the two sets of polynomials previously used.

Deletion of a subformula from an original input formula in QE first induces possible deletion of polynomials in retained projection bases associated with the deleted subformula. This requires a complex data structure for projection that allows for tracking of "reasons" for any one polynomial's existence (at a non trivial level). For example, one can only justify removal of a polynomial if, recursively, its existence can *only* be attributed to a polynomial to be removed via decrementality. This is important when cognizant of the fact that CAD projection sets are really set theoretic, and one should be careful not to delete, for example, a discriminant that is actually also equal to the resultant of two polynomials from the previous level. `QuantifierElimination` does not feature such a complex data structure, so where deletion is concerned, the projection bases remain the same. In other words the `projection` object is untouched. This also means we cannot "merge" `CADCell`s together, because while `CADCell`s have some "reasons" for existence via their local cell description, the polynomials from that description themselves do not have reasons for existence. Decremental CAD here reduces entirely to modification of the lifted CAD tree via traversal into continued regulation lifting. Hence, CAD subtrees may be removed due to changing truth values of cells in light of the modified input formula.

[45] describes a projection data structure that enables decrementality (backtracking) to the extent that projection polynomials can be removed on removal of constraints. The projection data structure forms a graph such that the aforementioned "reasons" are available for projection polynomials. The associated projection operator is the McCallum projection, and equational constraints are not considered. Equational constraints are essentially the frustration such that `QuantifierElimination` chooses a caching approach for incremental projection, while [45] can take a set arithmetic approach. Further, cells associate the polynomials that they were lifted with, to enable

more bespoke decrementality on cells than can be provided here. There is potential for `QuantifierElimination` to eventually support a more sophisticated projection data structure similar to [45] that would better accommodate decremental CAD. This would change the methods to be described in this section significantly, but enable individual calls to decremental CAD, and subsequent evolutionary operations to suffer less from "geometry bloat". In other words, evolutionary CAD can become more "destructive" instead of merely constructive, where the only "destruction" possible in decremental CAD at present is upon reevaluation of cells' truth values and the associated discarding of CAD subtrees. An example of a proactive destructive operation enabled by a more sophisticated data structure may include "merging" of existing cells together. Currently there is no analogy to a proactive destructive operation such as tree pruning in VTS (Section 5.1.4). Development of `QuantifierElimination` has mostly focused on constructive incremental CAD to enable poly-algorithmic QE.

We require two algorithms for decremental QE by CAD. As usual, the first, Algorithm 55, is a tree traversal algorithm acting root down. We perform deletion of a subformula at a specific atomic position from the structural formula of every `CADCell` traversed, such that cells evaluate the top level formula after identical deletion. Much of the logic is exactly the same as that of Algorithm 52 or 53 to deduce when to truncate a CAD subtree, or identify a cell that now has an indeterminate truth value where it did not previously. The logic on deduction of curtains from those algorithms does not need to appear here, because there are no new projection polynomials and hence no new pivots to check for additional curtains. As discussed above, truncation of the CAD tree is the only destructive operation, and there exists no proactive "merging" or similar in this algorithm.

Proposition 84. *Usage of Algorithm 48 modifies the CAD subtree rooted at c by deletion of the subformula at atomic position α from the structural (real) Tarski formula associated with each cell at that subtree. The containers* cad *and* leaves *are populated with unevaluated cells and meaningful leaf cells from the subtree of c respectively.*

Proof.

1. Much of the discussion from Algorithm 52 holds similarly due to the similar structure of the algorithms. This includes protection of truth values, treatment of unevaluated cells, and recursion to evaluate the repurposing formula are all once again relevant.

2. Checking for new Lazard curtains is irrelevant because there are guaranteed to be no new pivot polynomials to check, due to no new projection polynomials. Likewise, merging in of new geometry is seen to be irrelevant, hence there are no calls to Algorithm 51.

Proposition 85. *Usage of Algorithm 56 CADDecremental achieves the goal of decremental QE by Partial CAD, enabling deletion of a subformula at atomic position α from Φ for a past quantified QE problem $Q_{n-m+1}x_{n-m+1} \dots Q_n x_n \ \Phi$ donating* `CADData` data *from that elimination, such that we can produce QE output for the problem after this deletion.*

Proof. Considering `QuantifierElimination` has no methodology for "pruning" the projection bases due to removal of subformulae from the original formula, we rely entirely on usage of Lemma 77, because the past sets of polynomials fed through projection are guaranteed to be supersets of those required to lift to sign invariance for the top level formula after deletion. As usual, we move to "correct" the CAD tree such that it evaluates truth values for the top level formula after deletion via usage of Algorithm 55 to traverse the entire tree, and considering the tree may be insufficient to describe QE, proceed with regulation lifting into lifting failure recovery. The regulation lifting may be overly verbose when the projection polynomials are excessively numerous due to the lack of removal of such, and so the lifting is certainly merely "sufficient". Alternatively one relifts the tree from scratch before lifting failure recovery, again such an option controlled by the Maple keyword option `'Relift'`.

Algorithm 51 Incremental CAD Merge Part 2

```
27:                 if pastLB ≤ lb then
28:                     if pastUB < ub then
29:                         try
30:                             ( cell_right ↦ lowerbound, rtf ) ←
                                    refineIsolatingIntervals( cell_right ↦ lowerbound, rtf
                                    )
31:                             cell_left ↦ upperbound ← cell_right ↦ lowerbound
32:                             b′ ← the polynomial from rtf
33:                             ( lb, ub ) ← the interval from rtf
34:                             ( pastLB, pastUB ) ← the interval from
                                    cell_right ↦ lowerbound
35:                             if c ↦ children[i + 1] ↦ local_index mod 2 = 1 then
36:                                 cell_central ↦ lowerbound ← cell_right ↦ lowerbound
37:                             end if
38:                             i++      ▷ Because if the above succeeded, lb > pastUB
39:                         catch "Roots the same":
40:                             dontAdd ← true
41:                         end try
42:                     else
43:                         try
44:                             ( cell_right ↦ lowerbound, rtf ) ←
                                    refineIsolatingIntervals( cell_right ↦ lowerbound, rtf
                                    )
45:                             cell_left ↦ upperbound ← cell_right ↦ lowerbound
46:                             b′ ← the polynomial from rtf
47:                             ( lb, ub ) ← the interval from rtf
48:                             ( pastLB, pastUB ) ← the interval from
                                    cell_right ↦ lowerbound
49:                             if c ↦ children[i + 1] ↦ local_index mod 2 = 1 then
50:                                 cell_central ↦ lowerbound ← cell_right ↦ lowerbound
51:                             end if
52:                         catch "Roots the same":
53:                             dontAdd ← true
54:                         end try
55:                     end if
56:                 elseif pastLB ≤ ub then
57:                     try
58:                         ( rtf, cell_right ↦ lowerbound) ← refineIsolatingIntervals(
                                rtf, cell_right ↦ lowerbound )
59:                         cell_left ↦ upperbound ← cell_right ↦ lowerbound
60:                         b′ ← the polynomial from rtf
61:                         ( lb, ub ) ← the interval from rtf
62:                         ( pastLB, pastUB ) ← the interval from
                                cell_right ↦ lowerbound
```

Algorithm 51 Incremental CAD Merge Part 3

```
63:                         if c ↦ children[i + 1] ↦ local_index mod 2 = 1 then
64:                             cell_central ↦ lowerbound ← cell_right ↦ lowerbound
65:                         end if
66:                     catch "Roots the same":
67:                         dontAdd ← true
68:                     end try
69:                 end if
70:             end if
71:         until dontAdd or ( i = 1 or lb > self ↦ children[i]↦ lowerbound ) and
                                ( i > nc − pastInc or
                                ub > self ↦ children[i]↦ upperbound )
72:         if not dontAdd then
73:             mRtf ← RootOf( T[b′], index = real[j] )▷ Real algebraic function for
                    new root
74:             if sections then
75:                 newCell ← CADCell( x = mRtf, c, lvl, cell_left ↦ local_index + 1,
                        false, rtf )
76:                 Insert newCell at position i + 1 in c ↦ children   ▷ To keep child
                        cells sorted, we do this manually
77:             end if
78:             Let crRtf be the real algebraic function describing the upper bound
                    of cell_right
79:             if nc ≤ i then                    ▷ edge case, construct "rightmost" cell
80:                 if bounds were passed then
81:                     newCell ← CADCell( mRtf < x, c, lvl,
                            cell_left ↦ local_index + 2, false, rtf, bounds[2] )
82:                 else
83:                     newCell ← CADCell( mRtf < x, c, lvl,
                            cell_left ↦ local_index + 2, false, rtf, ∞ )
84:                 end if
85:             else
86:                 newCell ← CADCell( mRtf < x ∧ x < crRtf, c, lvl,
                        cell_left ↦ local_index + 2, false, rtf, cell_right ↦ lowerbound )
87:             end if
88:             replaceUpperBound( cell_left, rtf, mRtf )
89:             Insert newCell at position i + newInc in c ↦ children  ▷ To keep the
                    child cells c ↦ children sorted, we do this manually
90:             for k from i + newInc + 1 to |c ↦ children| do
91:                 c ↦ children[k] ↦ local_index += 2
92:             end for
93:         end if
94:     end for
95: end for
96: end procedure
```

Algorithm 52 Repurposing of a CAD Tree for a different formula

Input: self a `CADCell` to modify, cad a `QEContainer` for unevaluated cells, leaves a `QEContainer` for meaningful leaf cells, problemCells a `QEContainer` for cells with lifting failures, quants an `Array` of the quantifiers $Q_{n-m+1}, \ldots, Q_n$, vars an `Array` of the variables for the problem $x_1, \ldots, x_n$, m the number of quantifiers, n the number of variables, incBases an `Array` of bases of polynomials coprime to those used in the existing projection for this CAD, Ψ the new formula to repurpose this cell with, newOpen boolean flag dictating if incrementalCADMerge only builds "open" new geometry, newPivots an `Array` of boolean flags representing if a new pivot was introduced per every canonical CAD projection level, bases the `projection` object for the CAD

Output: No meaningful return, but self modified in place (and via traversal, the CAD subtree with root self modified in place)

1: **procedure** traverseCADTreeModify(self, cad, leaves, problemCells, quants, vars, m, n, incBases, Ψ, newOpen, newPivots, bases)
2: **if** self $\mapsto$ level $= 0$ **then**
3: self $\mapsto$ tarski_formula $\leftarrow \Psi$
4: Unassign self $\mapsto$ truth_value
5: **for** i **from** 2 **to** |self $\mapsto$ children| **do**
6: self $\mapsto$ children[i] $\mapsto$ truth_value $\leftarrow$ FAIL
7: **end for**
8: incrementalCADMerge(self, incBases[1], vars[1], 1, newOpen)
9: **for** c **in** self $\mapsto$ children **do**
10: traverseCADTreeModify(c, cad, leaves, problemCells, quants, vars, m, n, incBases, Ψ, newOpen)
11: **end for**
12: **elseif** self $\mapsto$ sample_point is unset **or** self $\mapsto$ tarski_formula is unset **then**
13: Unassign self $\mapsto$ truth_value
14: **try**
15: **if** evalAndSetTruthValue(self, leaves) = FAIL **then**
16: Add self to cad
17: **return**
18: **else**
19: **return**
20: **end if**
21: **catch** "Could not deduce sign":
22: Add [self,"Could not deduce sign"] to problemCells
23: **return**
24: **end try**
25: **else**
26: **if** $n > 1$ **and** self $\mapsto$ level $< n$ **and** newPivots[self $\mapsto$ level $+ 1$] **and** detectLazardCurtain(self, bases) **then**
27: Unevaluate self by discarding its truth_value, tarski_formula, tarski_formula_structural, children
28: Add [self,"Lazard curtain"] to problemCells
29: **return**

Algorithm 52 Repurposing of a CAD Tree, Part 2

```
30:         end if
31:         try
32:             self ↦ tarski_formula ← evaluateTFArrayAtSP( Ψ, self ↦ sample_point
                    )
33:         catch "Could not deduce sign":
34:             Add [self,"Could not deduce sign"] to problemCells
35:             return
36:         end try
37:         if self ↦ tarski_formula is a boolean value then
38:             self ↦ truth_value ← self ↦ tarski_formula
39:             Unassign self ↦ children
40:             Add self to leaves
41:             return
42:         else
43:             Unassign self ↦ truth_value
44:             if self ↦ children is an Array then
45:                 lvl ← self ↦ level + 1
46:                 x ← vars[lvl]
47:                 SP ← GetSamplePoints( self )              ▷ Full sample point of self
48:                 ( B, T ) ← univariateBasisAtLazard( incBases[lvl], c )
49:                 incrementalCADMerge( self, B, T, x, lvl, newOpen )
50:                 for i from 2 to |self ↦ children| do
51:                     self ↦ children[i] ↦ truth_value ← FAIL
52:                 end for
53:                 for c in self ↦ children do
54:                     traverseCADTreeModify( c, cad, leaves, problemCells, quants,
                            vars, m, n, incBases, self ↦ tarski_formula, newOpen )
55:                 end for
56:             else
57:                 Add self to cad
58:                 return
59:             end if
60:         end if
61:     end if
62:     PRPTV( self, quants, m, n, cad, leaves, problemCells )
63:     return
64: end procedure
```

Algorithm 53 Traversal of a CAD tree inserting a formula at a certain atomic position in the Tarski formula for every cell

Input: self a `CADCell` to perform formula insertion on, cad a `QEContainer` for unevaluated cells, leaves a `QEContainer` for leaf cells, problemCells a `QEContainer` for cells where lifting failed, quants an `Array` of the quantifiers $Q_{n-m+1}, \ldots, Q_n$, vars an `Array` of the variables for the problem $x_1, \ldots, x_n$, m the number of quantifiers, n the number of variables, incBases an `Array` of bases of polynomials coprime to those used in the existing projection for this CAD, operation, Ψ the formula at the top level to insert, atpos, newOpen, boolean flag representing whether incrementalCADMerge should build "open" new geometry only, newPivots an `Array` of boolean flags representing if a new pivot was introduced per every canonical CAD projection level, bases the `projection` object for the CAD

Output: No meaningful return, but self modified in place (and via traversal, the subtree with root self modified in place)

1: **procedure** TRAVERSECADTREEINSERT(self, cad, leaves, problemCells, quants, vars, m, n, incBases, operation, Ψ, atpos, newOpen, newPivots, bases)
2: **if** self $\mapsto$ level $= 0$ **then**
3: self $\mapsto$ tarski_formula $\leftarrow$ insertFormula(self $\mapsto$ tarski_formula, operation, Ψ, atpos)
4: Unassign self $\mapsto$ truth_value
5: **for** i **from** 2 **to** |self $\mapsto$ children| **do** ▷ Protection of truth values
6: self $\mapsto$ children[i] $\mapsto$ truth_value $\leftarrow$ FAIL
7: **end for**
8: incrementalCADMerge(self, incBases[1], vars[1], 1, newOpen)
9: **for** c **in** self $\mapsto$ children **do**
10: traverseCADTreeInsert(c, cad, leaves, problemCells, quants, vars, m, n, incBases, oper, Ψ, atpos, newOpen, newPivots, bases)
11: **end for**
12: **elseif** self $\mapsto$ sample_point is unset **or** self $\mapsto$ tarski_formula is unset **then**
13: Unassign self $\mapsto$ truth_value
14: **try**
15: **if** evalAndSetTruthValue(self, leaves) = FAIL **then**
16: Add self to cad
17: **return**
18: **else**
19: **return**
20: **end if**
21: **catch** "Could not deduce sign":
22: Add [self,"Could not deduce sign"] to problemCells
23: **return**
24: **end try**
25: **else**
26: **if** $n > 1$ **and** self $\mapsto$ level $< n$ **and** newPivots[self $\mapsto$ level $+ 1$] **and** detectLazardCurtain(self, bases) **then**

Algorithm 53 Traversal of a CAD tree to perform insertion of a new subformula, Part 2

```
27:             Unevaluate self by discarding its truth_value, tarski_formula,
                    tarski_formula_structural, children
28:             Add [self,"Lazard curtain"] to problemCells
29:             return
30:         end if
31:         Let α be self ↦ sample_point
32:         try
33:             subbed ← evaluateTFArrayAtSP( Ψ, α )
34:         catch "Could not deduce sign":
35:             Add [self,"Could not deduce sign"] to problemCells
36:             return
37:         end try
38:         self ↦ tarski_formula_structural ← insertFormula(
                self ↦ tarski_formula_structural, operation, atpos )
39:         if self ↦ tarski_formula is a boolean value then
40:             self ↦ truth_value ← self ↦ tarski_formula
41:             Unassign self ↦ children
42:             Add self to leaves
43:             return
44:         else
45:             Unassign self ↦ truth_value
46:             if self ↦ children is an Array then
47:                 lvl ← self ↦ level + 1
48:                 x ← vars[lvl]
49:                 SP ← GetSamplePoints( self )              ▷ Full sample point of self
50:                 ( B, T ) ← univariateBasisAtLazard( incBases[lvl], c )
51:                 incrementalCADMerge( self, B, T, x, lvl, newOpen )
52:                 for i from 2 to |self ↦ children| do      ▷ Protection of truth values
53:                     self ↦ children[i] ↦ truth_value ← FAIL
54:                 end for
55:                 for c in self ↦ children do
56:                     traverseCADTreeInsert( c, cad, leaves, problemCells, quants,
                            vars, m, n, incBases, oper, self ↦ tarski_formula, atpos,
                            newOpen, newPivots, bases )
57:                 end for
58:             else
59:                 Add self to cad
60:                 return
61:             end if
62:         end if
63:     end if
64:     PRPTV( self, quants, m, n, cad, leaves, problemCells )
65:     return
66: end procedure
```

Algorithm 54 Incremental QE by pure Partial CAD, via insertion of a new formula at a certain atomic position

Input: data `CADData` for a previous computation solving $Q_{n-m+1}x_{n-m+1}\dots Q_n x_n\ \Phi$, "oper" a boolean operator — either `And` or `Or`, α the atomic position to insert at, ψ the incoming formula to insert into Φ at atomic position α with operation oper, a myriad of CAD keyword options such as "UseGroebner", "UseEquations", "PropagateECs" related to projection, and Relift a boolean flag dictating if the user wants the CAD to be relifted, OpenCAD a boolean flag via keyword option representing if the user requests only sectors to be lifted

Output: General QE output from CAD obliging however many output arguments requested (anything up to and including witnesses, quantifier free output, and `CADData`)

1: **procedure** CADIncremental(data, oper, α, ψ, UseGroebner, UseEquations, PropagateECs, OpenCAD)
2: $k \leftarrow$ the number of new variables contained in ψ not contained in the variables for data ▷ Such variables are taken to be free variables
3: **if** $k > 0$ **then**
4: Sort the new variables $[y_1, \dots, y_k]$ from ψ by an appropriate heuristic such as the Brown heuristic or ECHeuristic using A and E, and prepend them to the past variable ordering data $\mapsto$ Vars $= [x_1, \dots, x_n]$ to receive vars
5: **end if**
6: Let data $\mapsto$ RootCell be the root cell for the past QE computation, and data $\mapsto$ Bases the `projection` object for the past computation
7: Insert Ψ at atomic position atpos with operation oper in data $\mapsto$ RootCell $\mapsto$ tarski_formula, inplace
8: (A, E) $\leftarrow$ getPolySets(data $\mapsto$ RootCell $\mapsto$ tarski_formula)
9: (incBases, newPivots) $\leftarrow$ projectionIncremental(data $\mapsto$ Bases, A, E, vars, $n + k$, 0, k, UseGroebner, PropagateECs, UseEquations) ▷ Algorithm 50
10: problemCells $\leftarrow$ an empty container
11: **if** Relift **or** $k > 0$ **then**
12: data $\mapsto$ RootCell $\leftarrow$ `CADCell`(data $\mapsto$ RootCell $\mapsto$ tarski_formula) ▷ Such that we receive a fresh root cell without child cells
13: cad $\leftarrow$ a container containing data $\mapsto$ RootCell
14: leaves $\leftarrow$ an empty container
15: localopen $\leftarrow$ OpenCAD **and** hasAllStrongRelations(data $\mapsto$ RootCell $\mapsto$ tarski_formula)
16: **else**
17: localopen $\leftarrow$ OpenCAD **and** hasAllStrongRelations(ψ)
18: **if not** localopen **and** Open CAD used before for data **then**
19: **ERROR** — the lifting may fail to deduce QE due to lack of sections (proof of Proposition 83)
20: **end if**

Algorithm 54 Incremental CAD by Pure Partial CAD, Part 2

21: traverseCADTreeInsert(data $\mapsto$ RootCell, cad, leaves, problemCells, $[Q_{n-m+1}, \ldots, Q_n]$, vars, m, $n+k$, incBases, oper, Ψ, atpos, newPivots, data $\mapsto$ Bases) ▷ Algorithm 53
22: **end if**
23: Code Fragment 28 (regulation Partial CAD lifting), applied in an evolutionary context to the relevant variables
24: Code Fragment 35 (lifting failure recovery by curtain decomposition), applied in an evolutionary context to the relevant variables
25: **return** QE output dependent on how many output arguments requested
26: **end procedure**

Algorithm 55 Traversal of a CAD tree to perform deletion of a formula at a particular atomic position from the Tarski formula held by each cell

Input: self a `CADCell` to perform formula insertion on, cad a `QEContainer` for unevaluated cells, leaves a `QEContainer` for leaf cells, problemCells a `QEContainer` for cells where lifting failed, quants an `Array` of the quantifiers $Q_{n-m+1}, \ldots, Q_n$, vars an `Array` of the variables for the problem $x_1, \ldots, x_n$, m the number of quantifiers, n the number of variables, atpos

Output: No meaningful return, but self modified in place (and via traversal, the subtree with root self modified in place)

1: **procedure** traverseCADTreeDelete(self, cad, leaves, problemCells, quants, vars, m, n, atpos)
2: **if** self $\mapsto$ level $= 0$ **then**
3: Delete the subformula at atomic position atpos from self $\mapsto$ tarski_formula
4: Unassign self $\mapsto$ truth_value
5: **for** i **from** 2 **to** |self $\mapsto$ children| **do** ▷ Protection of truth values
6: self $\mapsto$ children[i] $\mapsto$ truth_value $\leftarrow$ FAIL
7: **end for**
8: **for** c in self $\mapsto$ children **do**
9: traverseCADTreeDelete(c, cad, leaves, problemCells, quants, vars, m, n, atpos)
10: **end for**
11: **elseif** self $\mapsto$ sample_point is unset **or** self $\mapsto$ tarski_formula is unset **then**
12: Unassign self $\mapsto$ truth_value
13: **try**
14: **if** evalAndSetTruthValue(self, leaves) = FAIL **then**
15: Add self to cad
16: **return**
17: **else**
18: **return**
19: **end if**
20: **catch** "Could not deduce sign":
21: Add [self,"Could not deduce sign"] to problemCells
22: **return**
23: **end try**

Algorithm 55 Traversal of a CAD tree to perform deletion of a formula at a particular atomic position, Part 2

```
24:     else
25:         self ↦ tarski_formula_structural ← insertFormula(
                self ↦ tarski_formula_structural, operation, atpos )
26:         Delete the subformula at atomic position atpos from
                self ↦ tarski_formula_structural
27:         self ↦ tarski_formula ← simplify( self ↦ tarski_formula_structural )
28:         if self ↦ tarski_formula is a boolean value then
29:             self ↦ truth_value ← self ↦ tarski_formula
30:             Unassign self ↦ children
31:             Add self to leaves
32:             return
33:         else
34:             Unassign self ↦ truth_value
35:             if self ↦ children is an Array then
36:                 for i from 2 to |self ↦ children| do      ▷ Protection of truth values
37:                     self ↦ children[i] ↦ truth_value ← FAIL
38:                 end for
39:                 for c in self ↦ children do
40:                     traverseCADTreeDelete( c, cad, leaves, problemCells, quants,
                            vars, m, n, newOpen )
41:                 end for
42:             else
43:                 Add self to cad
44:                 return
45:             end if
46:         end if
47:     end if
48:     PRPTV( self, quants, m, n, cad, leaves, problemCells )
49:     return
50: end procedure
```

Algorithm 56 Decremental QE by Partial CAD, by deletion of a formula at a certain atomic position

Input: data the `CADData` from a previous QE computation by Partial CAD for $Q_{n-m+1}x_{n-m+1}\ldots Q_n x_n\ \Phi(x_1,\ldots,x_n)$, α the atomic position of the subformula to delete from Φ, "Relift" a boolean flag dictating if the user requests the CAD tree to be relifted, "OpenCAD" a boolean flag via keyword option representing if the user requests only sectors to be lifted

Output: General QE output from CAD obliging however many output arguments requested (anything up to and including witnesses, quantifier free output, and `CADData`)

1: **procedure** CADDecremental(data, α, Relift, OpenCAD)
2: Let data $\mapsto$ RootCell be the root cell for the previous QE computation
3: problemCells $\leftarrow$ an empty container
4: **if** Relift **then**
5: $\Psi \leftarrow$ the formula resulting from deletion of the subformula at atomic position α from data $\mapsto$ RootCell $\mapsto$ tarski_formula
6: data $\mapsto$ RootCell `CADCell`(Ψ)
7: cad $\leftarrow$ a container containing data $\mapsto$ RootCell
8: leaves $\leftarrow$ an empty container
9: **else**
10: cad $\leftarrow$ an empty container
11: leaves $\leftarrow$ an empty container
12: traverseCADTreeDelete(data $\mapsto$ RootCell, cad, leaves, problemCells, $[Q_{n-m+1},\ldots,Q_n]$, $[x_1,\ldots,x_n]$, m, n, α) ▷ Algorithm 55, and the deletion at the root cell happens in first call
13: **end if**
14: localopen $\leftarrow$ OpenCAD **and** hasAllStrongRelations(data $\mapsto$ RootCell $\mapsto$ tarski_formula)
15: Code Fragment 28 (regulation Partial CAD lifting), applied in an evolutionary context to data
16: Code Fragment 35 (lifting failure recovery by curtain decomposition), applied in an evolutionary context to data
17: **return** QE output dependent on how many output arguments requested
18: **end procedure**

Chapter 6

Other Features of the Software

A software demo demonstrating many of the features mentioned in this section, including working with `CADData`, and the subpackage `QuantifierTools` can be found at [67].

6.1 The Subpackage `QuantifierTools`

`QuantifierTools` is a Maple subpackage of `QuantifierElimination` intended to make usage of QE more tractable, especially in terms of pedagogy. It attempts to extend Maple's status as a mathematical learning tool and toolbox. It features several functions that act on, manipulate, or produce Tarski-like formulae (i.e. rational or real Tarski formulae via Definitions 6 or 32):

- ConvertRationalConstraintsToTarski,
- ConvertToPrenexForm,
- AlphaConvert,
- GetAllPolynomials,
- NegateFormula,
- GetUnquantifiedFormula,
- GetEquationalConstraints,
- and SuggestCADOptions.

ConvertToPrenexForm will convert any rational Tarski formula expressible via Maple's inert operators to prenex form in the spirit of Definition 4, e.g. $\forall x\ P(x) \wedge \forall x\ Q(x) \mapsto \forall x \forall x_1\ P(x) \wedge Q(x_1)$. Prenex conversion may involve alpha conversion to resolve conflicts between variables. Pure alpha conversion without the prenex conversion of moving of quantifiers is implemented via AlphaConvert, under which $\forall x\ P(x) \wedge \forall x\ Q(x) \mapsto \forall x\ P(x) \wedge \forall x_1\ Q(x_1)$, hence renaming one of the variables. New

variables introduced in this way are generally indexed versions of those variables that had conflicts. GetAllPolynomials will return a set or list of the polynomials from all the constraints in a rational Tarski formula, for the purposes of passing to functions performing full CAD such as `QuantifierElimination`'s CylindricalAlgebraicDecompose, or the similarly named function below `RegularChains`. GetUnquantifiedFormula will remove all quantifiers from any rational Tarski formula. NegateFormula performs negation of a rational Tarski formula, pushing all instances of negation to the leaves of the expression (that is distributing $\neg$ via De Morgan's laws, such that $\neg$ does not manifest in the output).

ConvertRationalConstraintsToTarski is a function that will convert rational functions (of polynomials) within constraints to the equivalent Tarski formulae, i.e. taking formulae from Definition 6 to Definition 32. As many problems may arise as Rational Tarski formulae rather than (real) Tarski formulae, this is intended to encourage conversion of such problems to the conventional Tarski framework such that they are amenable as input to QE packages such as `QuantifierElimination`. Conversion of rational constraints to equivalent Tarski formulae will result in formulae on the numerators and denominators of such rational constraints, especially conditions of the sign of the denominators. For example

$$\frac{x}{y} < 0 \mapsto x < 0 \wedge y > 0 \vee x > 0 \wedge y < 0$$

where it is clear that on the right hand side y being identically 0 is excluded, which would result in a singularity on the left hand side.

GetEquationalConstraints will return a set of polynomials of all logically implied equational constraints from an input formula able to be deduced by `QuantifierElimination` (Section 3.7). Such a returned set coincides entirely with what `QuantifierElimination` would identify as equational constraints for projection with equational constraint optimisations — in other words, the set E from Algorithm 21 is returned by this function.

SuggestCADOptions is a function to easily realise the performance increases associated with the lifting related optimisations of Sections 3.5 and 3.6 (OpenCAD and lifting constraints). By examining a (quantified or unquantified) real Tarski formula intended for CAD input, SuggestCADOptions can provide a recommended sequence of arguments to pass to a `QuantifierElimination` CAD function such that lifting is less costly, omitting cells with less meaning for the associated problem. SuggestCADOptions provides the recommended values for the keyword options `'LiftingConstraints'` and `'OpenCAD'`, which are keyword options for all top level CAD procedures offered by `QuantifierElimination`, such as CylindricalAlgebraicDecompose or PartialCylindricalAlgebraicDecompose. Of course the first argument to return by SuggestCADOptions is the input formula, potentially modified as a result of extraction of lifting constraints as to avoid their use in projection. The remainder of the returned sequence is keyword options for `'LiftingConstraints'` and `'OpenCAD'`. The accepted values for `'LiftingConstraints'` are a set of lifting constraints (Definition 43), or the symbol `'positive'`, meaning "build in $\mathbb{R}^n_+$", which is not explicitly ever returned as part of output of SuggestCADOptions, in lieu of the equivalent $\{x_1 > 0, \ldots, x_n > 0\}$. By

default the value of `'LiftingConstraints'` is the empty set. The accepted value for `'OpenCAD'` is either *true* or *false*, and is by default *false*. If passed a conjunction without universal quantifiers, SuggestCADOptions will extract all lifting constraints from the formula as an appropriate Maple set to pass for `'LiftingConstraints'` (hence the output conjunction has those necessarily top level constraints removed). Meanwhile, it uses Algorithm 19 on the unquantified part of the formula to deduce if Open CAD would be appropriate due to lack of any weak relations providing rationale for production of sections. If it is, `'OpenCAD'` = *true* appears in the returned sequence, else we see `'OpenCAD'` = *false*. Figure 6-1 demonstrates usage of SuggestCADOptions producing non trivial output for relevant examples from this project's QE example database in Maple. SuggestCADOptions allows for easy usage of the OpenCAD and lifting constraint optimisations for users without insisting on usage of their potentially "destructive" CAD properties by default, because usage of either option in a non trivial way may omit production of cells of the CAD that may be desired for inspection by the user via `CADData`. Usage of SuggestCADOptions does not in itself deduce if extracted lifting constraints from a formula are intrinsically unsatisfiable (i.e. *false*) — this deduction lies with the parsing of lifting constraints by manageLiftingConstraints (Algorithm 20) within the ensuing CAD call.

6.2 Other Features

Querying properties of CAD via `CADData`

Being largely object oriented, `QuantifierElimination` allows for querying of `CADData` and `CADCell` objects for various properties. One may:

- Request the cell containing a point via GetCellContainingPoint to return a `CADCell` object to query, or simply its description as an extended Tarski formula via GetCellDescriptionContainingPoint. The cell may not be unique if the point is under-specified, i.e. has fewer than n coordinates. It may not be possible to find a satisfactory cell if the CAD omits cells due to usage of lifting constraints or OpenCAD.

- Use GetAllLeafCells to acquire a list of all leaf cells from a CAD, potentially per truth value via passing e.g. `'TruthValue'` = *true*. The function NumberOfLeafCells enumerates the number of leaf cells in the CAD, and also accepts the keyword option `'TruthValue'` to only enumerate the number of leaf cells with a specific truth value. The keyword option `'TruthValue'` is only intelligible whenever truth values were relevant for the CAD (the `CADData` must have been generated from usage of PartialCylindricalAlgebraicDecompose).

- Query the sign of a polynomial on a cell via SignOfPolynomialOnCell. Any cell is only guaranteed sign invariant on polynomials from the `CADData`'s projection bases, so the polynomial passed must factor into constituents from said bases for this function to be correct. We can check that the factors are contained in the projection bases more efficiently by only checking the bases of the same polynomial level as each factor.

```
> expr := QEExamples[ 'Collision' ];
```

$$expr := \exists y \exists x \exists t \ \frac{(x-t)^2}{4} + (y-10)^2 - 1 = 0 \wedge \frac{(-at+x)^2}{4} + (-at+y)^2 - 1 = 0 \wedge 0 < t \wedge 0 < a$$

```
> QuantifierTools:-SuggestCADOptions( expr );
```

$\exists y \exists x \exists t \ t^2 - 2xt + x^2 + 4y^2 - 80y + 396 = 0 \wedge 5a^2t^2 - 2atx - 8aty + x^2 + 4y^2 - 4 = 0$, $LiftingConstraints = \{-a < 0, -t < 0\}$, $OpenCAD = false$

```
> alias( R1 = RootOf( 5*_Z^2 - 12*_Z + 6, 70/10 .. 72/10 ) ):
> alias( R2 = RootOf( 5*_Z^2 - 12*_Z + 6, 168/10 .. 170/10 ) ):
> PartialCylindricalAlgebraicDecompose( expr );
```

$$a = R1 \vee R1 < a \wedge a < \frac{5}{7} \vee a = \frac{5}{7} \vee \frac{5}{7} < a \wedge a < 1 \vee a = 1 \vee 1 < a \wedge a < \frac{5}{3} \vee a = \frac{5}{3} \vee \frac{5}{3} < a \wedge a < R2 \vee a = R2$$

```
> expr := QEExamples[ 'Ball and Circular Cylinder' ];
```

$$expr := \exists z \exists x \exists y \ x^2 + y^2 + z^2 < 1 \wedge x^2 + (y + z - 2)^2 < 1$$

```
> QuantifierTools:-SuggestCADOptions( expr );
```

$$\exists z \exists x \exists y \ x^2 + y^2 + z^2 < 1 \wedge x^2 + y^2 + 2yz + z^2 - 4y - 4z < -3,\ LiftingConstraints = \{\},\ OpenCAD = true$$

```
> PartialCylindricalAlgebraicDecompose( % );
```

$$true$$

Figure 6-1: Usage of SuggestCADOptions to deduce suggested keyword option arguments associated with lifting optimisations (OpenCAD, Section 3.5, lifting constraints, Section 3.6) to pass to PartialCylindricalAlgebraicDecompose for quantified formulae. The RootOfs $R1$ and $R2$ represent distinct roots of the same polynomial — their isolating intervals are truncated to low precision for brevity.

- Print the full projection bases for a CAD's `CADData` via PrintProjection, including with highlighting of equational constraints and pivot sets by passing the keyword option `'verbose'` = *true*.
- Examine all the cell descriptions for a CAD for a full description of the geometry formed by the polynomials (`map(GetFullDescription,GetAllLeafCells(C))` where C is the `CADData` for the CAD). Via Section 3.4, the descriptions are particular extended Tarski formulae. Real algebraic numbers appear over real algebraic functions where possible in these descriptions.

This concludes a non exhaustive discussion of features which may even be extended by further development of `QuantifierElimination`. These features intend to add to `QuantifierElimination`'s goal to provide intuitive input and rich output.

Chapter 7

Benchmarking, Examples, and Comparisons to Other Software

7.1 Example Databases

Members of the $\mathcal{SC}^2$, and Computer Algebra community have been keen to have examples for QE that arise from real world examples, or at the very least examples that aren't "cherry-picked" and invented for the purpose of benchmarking. The use of randomly generated examples can also be contentious. Part of this project offers a Maple script compiling two databases of examples for Quantifier Elimination. These examples are built in the syntax of Maple, and in particular in the syntax of `QuantifierElimination`. Much of this database is adapted from [72], which was originally a database of examples for pure CAD (with many examples being QE examples stripped of their quantifiers). This is referred to further as "the QE database". Further examples are added to the database from mechanics applications in [39], and other examples from real algebraic geometry via [26, 58]. A last example added from real algebraic geometry is the Sharir cube [76]. This is a shape that can be modelled in space by polynomial constraints. Existence of a point within the shape implies its existence at all.

An additional database of economics examples is adapted for Maple (with formatting intelligible for `QuantifierElimination`) from [53], and offers a function that can pose the appropriate QE problems formed by "assumptions" A and "hypotheses" H in the spirit of [54]. For example, these can be fully universally quantified "theorems" $\forall \mathbf{x}\ A(\mathbf{x}) \Rightarrow H(\mathbf{x})$, or fully existentially quantified "examples" $\exists \mathbf{x}\ A(\mathbf{x}) \wedge H(\mathbf{x})$ or "counterexamples" $\exists \mathbf{x}\ A(\mathbf{x}) \wedge \neg H(\mathbf{x})$. The formulae themselves are largely linear or at worst quadratic. The existentially quantified "examples" are referred to as "the Economics database", and feature in QE benchmarking.

Lastly, this work also makes use of the example database [34], attributed to `SyNRAC` and its developers, referred to as the "SyNRAC database", featuring QE examples that can be parsed by Maple in a format intelligible to the `SyNRAC` package. Other QE example databases such as Remis [33] exist. Here we only use examples originally written/stored in Maple (Remis, for example, is for `Redlog`).

The retained CAD example database adapted from [72] is all unquantified, so is only relevant for benchmarking CAD. The CAD benchmarks also incorporate the QE examples from the QE and SyNRAC example databases, where they are always treated as unquantified to enable full scope for variable strategy. This database is referred to further as "the CAD database". The CAD benchmarks do not use the economics database, as these examples are largely highly numerous in the number of variables, yielding fairly poor data for full CAD. QE benchmarking incorporates examples from the QE, SyNRAC, and Economics example databases.

There is a repository of data [65] associated with the project, containing:

- The example databases contributed from the project, as files that can be read into Maple defining tables of examples, and associated functions to build or examine various examples,
- A `.pdf` file providing references for all examples from the example databases, and typesetting of the examples as associated QE problems,
- The benchmarking data produced from the benchmarking of this section as `.csv`s, and the Excel workbooks as `.xlsx` files processing said data into survival plots for the figures of this section,
- Copies of the survival plots themselves as `.png` files,
- The `bash` and Maple scripts used to generate the raw benchmarking data,
- Maple files providing auxiliary tools allowing for conversion of QE formulae between formats (such as that of `SyNRAC`, `RegularChains`, `QuantifierElimination`, and `QEPCAD B`),
- Maple worksheets (and corresponding identical `.pdf`s of the output) used in software demos to demonstrate features of `QuantifierElimination` at conferences and other events (a video of the latest software demo is available via [67]),
- Some `.pdf` files generated from Maple demonstrating case studies on Lazard curtains generated from an early development build of `QuantifierElimination` (matching those from Section 7.2),
- A `README` file to support the repository and explain the purpose of the aforementioned files further.This `README` also provides contact details for the author and information about obtaining the `QuantifierElimination` package before its official release.

7.2 Case Studies on Lazard Curtains

The following are case studies of occurrences of Lazard curtains (see Section 3.7.2) on various examples from the QE example databases used here. Full CAD (via CylindricalAlgebraicDecompose) is used on each example treated as an unquantified formula, with the variable ordering inherited from usage of ECHeuristic. Multiple equational

constraints are used via passing the keyword option `'UseEquations'` = multiple unless otherwise stated. All other keyword options, including those related to propagation of ECs and usage of Gröbner bases are as default unless otherwise stated. The case studies below were generated with the monomial ordering $\texttt{prod}(\texttt{tdeg}(H(x_1, \ldots, x_{n-\min(n,k)})), \texttt{plex}(x_n, \ldots, x_{n-\min(n,k)+1}))$ for Gröbner bases in an early development build of `QuantifierElimination` as opposed to the ordering under symmetry as suggested in Section 3.7.3. However, the discussion about curtains is entirely representative of the presented projection bases with respect to the associated variable orderings, and the case studies below could be induced via passing `'UseGroebner'` = *false* with appropriately modified equational constraints within each formula (as can be produced via GB with the given monomial ordering) when using the latest version of `QuantifierElimination`. The accompanying `.pdf` files in the supporting repository [65] demonstrate the methodology to generate the data from Maple with `QuantifierElimination`. Here, we are interested in the distinctions between point curtains and non point curtains, and to what extent usage of Algorithm 30 decomposeCurtainCellsCAD introduced by [56] allows us to recover from point and non point curtains.

A Real Implicitization Problem

$$\exists u \, \exists v \;\; -uv + x = 0 \wedge -uv^2 + y = 0 \wedge -u^2 + z = 0$$

For this 5 variable problem, the ordering chosen by ECHeuristic is $[u, v, y, z, x]$.

Multiple Equational Constraints

The full projection bases are:

$$\{vx - y, uv - x, \mathbf{ux - vz}\}$$
$$\{-v^2z + uy, \mathbf{u^2 - z}\}$$
$$\{\mathbf{uv^2 - y}\}$$
$$\{v\}$$
$$\{u\}$$

with equational constraints chosen as pivot highlighted in bold (all the polynomials at level 5 are the result of Gröbner bases on the original polynomials, which were all ECs). 15 cells with curtains have been collected by the end of standard lifting. The curtain cells are visualised as Table 7.1. None are identified as point curtains in Algorithm 30, which reconciles with the fact that every cell in Table 7.1 has a neighbour cell also existing in the table in terms of a perturbed index.

The set of polynomials treated as inequalities at the top level in projection is $\{vx - y, uv - x\}$, which is hence the set B_A as input in Algorithm 31. The corresponding sets resulting from full Lazard projection on this set are:

Cell Description	Sample Point	Cell Index
$u = 0 \wedge v = 0 \wedge y = 0 \wedge z < 0$	$[u = 0, v = 0, y = 0, z = -1]$	$[2, 2, 2, 1]$
$u = 0 \wedge v = 0 \wedge y = 0 \wedge 0 < z$	$[u = 0, v = 0, y = 0, z = 1]$	$[2, 2, 2, 3]$
$u = 0 \wedge v = 0 \wedge y = 0 \wedge z = 0$	$[u = 0, v = 0, y = 0, z = 0]$	$[2, 2, 2, 2]$
$u = 0 \wedge v < 0 \wedge y < 0 \wedge z = 0$	$[u = 0, v = -1, y = -1, z = 0]$	$[2, 1, 1, 2]$
$u = 0 \wedge v < 0 \wedge y = 0 \wedge z = 0$	$[u = 0, v = -1, y = 0, z = 0]$	$[2, 1, 2, 2]$
$u = 0 \wedge v < 0 \wedge 0 < y \wedge z = 0$	$[u = 0, v = -1, y = 1, z = 0]$	$[2, 1, 3, 2]$
$u = 0 \wedge v = 0 \wedge y < 0 \wedge z = 0$	$[u = 0, v = 0, y = -1, z = 0]$	$[2, 2, 1, 2]$
$u = 0 \wedge v = 0 \wedge y < 0 \wedge z < 0$	$[u = 0, v = 0, y = -1, z = -1]$	$[2, 2, 1, 1]$
$u = 0 \wedge v = 0 \wedge y < 0 \wedge 0 < z$	$[u = 0, v = 0, y = -1, z = 1]$	$[2, 2, 1, 3]$
$u = 0 \wedge v = 0 \wedge 0 < y \wedge z < 0$	$[u = 0, v = 0, y = 1, z = -1]$	$[2, 2, 3, 1]$
$u = 0 \wedge v = 0 \wedge 0 < y \wedge z = 0$	$[u = 0, v = 0, y = 1, z = 0]$	$[2, 2, 3, 2]$
$u = 0 \wedge v = 0 \wedge 0 < y \wedge 0 < z$	$[u = 0, v = 0, y = 1, z = 1]$	$[2, 2, 3, 3]$
$u = 0 \wedge 0 < v \wedge y < 0 \wedge z = 0$	$[u = 0, v = 1, y = -1, z = 0]$	$[2, 3, 1, 2]$
$u = 0 \wedge 0 < v \wedge y = 0 \wedge z = 0$	$[u = 0, v = 1, y = 0, z = 0]$	$[2, 3, 2, 2]$
$u = 0 \wedge 0 < v \wedge 0 < y \wedge z = 0$	$[u = 0, v = 1, y = 1, z = 0]$	$[2, 3, 3, 2]$

Table 7.1: Non-point curtains generated for 'A Real Implicitization Problem'.

$$\{vx - y, uv - x\}$$
$$\emptyset,$$
$$\{y, uv^2 - y\}$$
$$\{v\}$$
$$\{u\}$$

In particular, the polynomials at level 1 completely coincide, with the innermost recursion on Algorithm 31 taking a set difference to receive the empty set to isolate roots of, hence building no new geometry as a result. After lifting the 15 cells with curtains normally in the lifting loop from Algorithm 30, we receive a total of 591 leaf cells.

Single Equational Constraint

The projection bases produced with usage of just a single equational constraint in x allowed by passing `'UseEquations' = 'single'` are:

$$\{vx - y, uv - x, \mathbf{ux - vz}\}$$
$$\{-v^2z + uy, u^2 - z\}$$
$$\{y, uv^2 - y\}$$
$$\{v\}$$
$$\{u\}$$

where one notes an extra contribution of y because of the addition of the trailing coefficient of $-v^2z + uy$ in z. Hence we receive a total of 951 leaf cells, roughly double of that of the case with multiple equational constraints due to y splitting the real line

in half in y. The situation in terms of curtain cells however remains exactly the same as that for multiple equational constraints.

Ellipse A

$\exists y\, \exists x\, x^2+y^2-1 = 0 \wedge b^2(x-c)^2+a^2y^2-a^2b^2 = 0 \wedge 0 < a \wedge a < 1 \wedge 0 < b \wedge b < 1 \wedge 0 \leq c \wedge c < 1$

For this 5 variable problem, the ordering chosen by ECHeuristic is $[y, x, c, b, a]$.

Multiple Equational Constraints

The full projection bases are:

$$\{a, a-1, \mathbf{b^2(x-c)^2 + a^2y^2 - a^2b^2}\}$$
$$\{b, b-1, b-y, y+b, b^2c^2 - b^2cx - b^2y^2 + y^2\}$$
$$\{c, c-1, c-2x, cr - 2cx - y^2, c^2 - 2cx - y^2 + 1\}$$
$$\{x, 2x-1, x^2+y^2, y^2+2x-1, y^2+2x-2, \mathbf{x^2 + y^2 - 1}\}$$
$$\{y, y-1, y+1, y^2+3, 4y^2-3\}$$

with equational constraints chosen as pivot in bold. This problem is not purely of equations. The Gröbner basis changes nothing for the equational constraints. 277 curtain cells that are not point curtains are obtained via standard lifting on this full projection. They are far too numerous to list in full here, but a sample of them are visualised in Table 7.2.

There are an additional 36 point curtains identified in Algorithm 30 via the criteria on neighbour cells. Again, a sample of them are visualised in Table 7.3.

The basis of inequalities B_A in Algorithm 30 is this time $\{a, a-1\}$. They are univariate in the last variable a, hence projection on this set is trivial, and none of the calls on Algorithm 31 receive any polynomials at all to build geometry around. Having passed this, after lifting the cells with curtains as standard we receive a total of 92233 leaf cells.

Single Equational Constraint

With just one single equational constraint in a being used due to passing of the keyword option `'UseEquations' = 'single'`, the projection sets become:

$$\{a, a-1, \mathbf{b^2(x-c)^2 + a^2y^2 - a^2b^2}\}$$
$$\{b, b-1, b-y, y+b, b^2c^2 - b^2cx - b^2y^2 + y^2\}$$
$$\{c, c-1, c-2x, cr - 2cx - y^2, c^2 - 2cx - y^2 + 1\}$$
$$\{x, 2x-1, x^2+y^2, y^2+2x-1, y^2+2x-2, x^2+y^2-1\}$$
$$\{y, y-1, y+1, y^2+3, y^2-2, 4y^2+1, 4y^2-3, y^2+1, y^2-2y+2, y^2+2y+2\}$$

Cell Description	Sample Point	Cell Index
$1 < y \wedge \frac{1}{2} < x \wedge c = \text{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \text{index} = \text{real}_1\right) \wedge b = \text{RootOf}\left(\_Z^2 - y^2, \text{index} = \text{real}_2\right)$	$[y = 2, x = 2, c = \text{RootOf}\left(\_Z^2 - 4\,\_Z - 3, -\frac{1163282145553801 7}{18014398509481984}.. -\frac{23265642911075953}{36028797018963968}\right), b = 2]$	$[11, 9, 4, 8]$
$1 < y \wedge x = \frac{1}{2} \wedge c = \text{RootOf}\left(\_Z^2 - y^2 - \_Z + 1, \text{index} = \text{real}_1\right) \wedge b = \text{RootOf}\left(\_Z^2 - y^2, \text{index} = \text{real}_2\right)$	$[y = 2, x = \frac{1}{2}, c = \text{RootOf}\left(\_Z^2 - \_Z - 3, -\frac{37296678482 81}{22918671393097}.. -\frac{22918671393097}{45837342786194}\right), b = 2]$	$[11, 8, 4, 8]$
$1 < y \wedge x = \frac{1}{2} \wedge c = \text{RootOf}\left(\_Z^2 - y^2 - \_Z + 1, \text{index} = \text{real}_1\right) \wedge b = \text{RootOf}\left(\_Z^2 - y^2, \text{index} = \text{real}_1\right)$	$[y = 2, x = \frac{1}{2}, c = \text{RootOf}\left(\_Z^2 - \_Z - 3, -\frac{37296678482 81}{22918671393097}\right), b = -2]$	$[11, 8, 4, 2]$
$1 < y \wedge 0 < x \wedge x < \frac{1}{2} \wedge c = \text{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \text{index} = \text{real}_1\right) \wedge b = \text{RootOf}\left(\_Z^2 - y^2, \text{index} = \text{real}_2\right)$	$[y = 2, x = \frac{1}{3}, c = \text{RootOf}\left(3\,\_Z^2 - 2\,\_Z - 9, -\frac{844418672662085 05263}{59029581035870565 1712}.. -\frac{16888373453241701 16499}{11805916207174113 03424}\right), b = 2]$	$[11, 7, 4, 8]$
$1 < y \wedge 0 < x \wedge x < \frac{1}{2} \wedge c = \text{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \text{index} = \text{real}_1\right) \wedge b = \text{RootOf}\left(\_Z^2 - y^2, \text{index} = \text{real}_1\right)$	$[y = 2, x = \frac{1}{3}, c = \text{RootOf}\left(3\,\_Z^2 - 2\,\_Z - 9, -\frac{844418672662085 05263}{59029581035870565 1712}.. -\frac{16888373453241701 16499}{11805916207174113 03424}\right), b = -2]$	$[11, 7, 4, 2]$

Table 7.2: Sample of five of the non point curtains generated for ‘Ellipse A’.

Cell Description	Sample Point	Cell Index
$1 < y \wedge \frac{1}{2} < x \wedge c = \mathrm{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(\_Z^2 - y^2, \mathrm{index} = \mathrm{real}_2\right)$	$[y = 2, x = 2, c = \mathrm{RootOf}\left(\_Z^2 - 4\,\_Z - 3, \frac{3427959418612364645577}{7378697629483820646 4}..\frac{85698985465309116151}{18446744073709551616}\right), b = 2]$	$[11, 9, 12, 8]$
$\mathrm{RootOf}\left(4\,\_Z^2 - 3, \frac{255605583759080241621}{295147905179352825856}..\frac{511211167518160483269}{590295810358705651712}\right) < y \wedge y < 1 \wedge -\frac{y^2}{2} + 1 < x \wedge c = \mathrm{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(\_Z^2 - y^2, \mathrm{index} = \mathrm{real}_1\right)$	$[y = \frac{8}{9}, x = 2, c = \mathrm{RootOf}\left(81\,\_Z^2 - 324\,\_Z + 17, \frac{23297936365258988003 01}{590295810358705651712}..\frac{29122420456573735004 1}{73786976294838206464}\right), b = -\frac{8}{9}]$	$[9, 13, 10, 2]$
$0 < y \wedge y < \mathrm{RootOf}\left(4\,\_Z^2 - 3, \frac{255605583759080241621}{295147905179352825856}..\frac{511211167518160483269}{590295810358705651712}\right) \wedge -\frac{y^2}{2} + 1 < x \wedge c = \mathrm{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(\_Z^2 - y^2, \mathrm{index} = \mathrm{real}_1\right)$	$[y = \frac{1}{2}, x = 2, c = \mathrm{RootOf}\left(4\,\_Z^2 - 16\,\_Z + 3, \frac{274018863121014207}{72057594037927936}..\frac{68504715780253815}{18014398509481984}\right), b = -\frac{1}{2}]$	$[7, 13, 10, 2]$
$y = \mathrm{RootOf}\left(4\,\_Z^2 - 3, -\frac{511211167518160483251}{590295810358705651712}.. -\frac{408968934014528386598 1}{472236648286964521369 6}\right) \wedge -\frac{\left(\mathrm{RootOf}\left(4\,\_Z^2 - 3, -\frac{511211167518160483251}{590295810358705651712}.. -\frac{408968934014528386598 1}{472236648286964521369 6}\right)\right)^2}{2} + 1 < x \wedge c = \mathrm{RootOf}\left(4\,\_Z^2 - 8\,\_Z\,x + 1, \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(4\,\_Z^2 - 3, \mathrm{index} = \mathrm{real}_2\right)$	$[y = \mathrm{RootOf}\left(4\,\_Z^2 - 3, -\frac{511211167518160483251}{590295810358705651712}.. -\frac{408968934014528386598 1}{472236648286964521369 6}\right), x = 2, c = \mathrm{RootOf}\left(4\,\_Z^2 - 16\,\_Z + 1, \frac{17312873468883}{4398046511104}..\frac{138502987751091}{35184372088832}\right), b = -\mathrm{RootOf}\left(4\,\_Z^2 - 3, -\frac{511211167518160483251}{590295810358705651712}.. -\frac{408968934014528386598 1}{472236648286964521369 6}\right)]$	$[4, 11, 10, 6]$
$-1 < y \wedge y < \mathrm{RootOf}\left(4\,\_Z^2 - 3, -\frac{511211167518160483251}{590295810358705651712}.. -\frac{408968934014528386598 1}{472236648286964521369 6}\right) \wedge -\frac{y^2}{2} + 1 < x \wedge c = \mathrm{RootOf}\left(\_Z^2 - 2\,\_Z\,x - y^2 + 1, \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(\_Z^2 - y^2, \mathrm{index} = \mathrm{real}_2\right)$	$[y = -\frac{8}{9}, x = 2, c = \mathrm{RootOf}\left(81\,\_Z^2 - 324\,\_Z + 17, \frac{23297936365258988003 01}{590295810358705651712}..\frac{29122420456573735004 1}{73786976294838206464}\right), b = \frac{8}{9}]$	$[3, 13, 10, 6]$
$y = -1 \wedge \frac{1}{2} < x \wedge c = \mathrm{RootOf}\left(\_Z\,(\_Z - 2\,x), \mathrm{index} = \mathrm{real}_2\right) \wedge b = \mathrm{RootOf}\left(\_Z^2 - 1, \mathrm{index} = \mathrm{real}_2\right)$	$[y = -1, x = 2, c = 4, b = 1]$	$[2, 5, 8, 6]$

Table 7.3: Sample of five of the point curtains generated for ‘Ellipse A’.

Cell Description	Sample Point	Cell Index
$t = 0 \wedge 0 < s \wedge r < -s \wedge b = 0$	$[t = 0, s = 1, r = -2, b = 0]$	$[2, 3, 1, 2]$
$t = 0 \wedge 0 < s \wedge r = -s \wedge b = 0$	$[t = 0, s = 1, r = -1, b = 0]$	$[2, 3, 2, 2]$
$t = 0 \wedge 0 < s \wedge -s < r \wedge b = 0$	$[t = 0, s = 1, r = 0, b = 0]$	$[2, 3, 3, 2]$
$t = 0 \wedge s = 0 \wedge 0 < r \wedge b = 0$	$[t = 0, s = 0, r = 1, b = 0]$	$[2, 2, 3, 2]$
$t = 0 \wedge s = 0 \wedge r = 0 \wedge b = 0$	$[t = 0, s = 0, r = 0, b = 0]$	$[2, 2, 2, 2]$
$t = 0 \wedge s = 0 \wedge r < 0 \wedge b = 0$	$[t = 0, s = 0, r = -1, b = 0]$	$[2, 2, 1, 2]$
$t = 0 \wedge s < 0 \wedge -s < r \wedge b = 0$	$[t = 0, s = -1, r = 2, b = 0]$	$[2, 1, 3, 2]$
$t = 0 \wedge s < 0 \wedge r = -s \wedge b = 0$	$[t = 0, s = -1, r = 1, b = 0]$	$[2, 1, 2, 2]$
$t = 0 \wedge s < 0 \wedge r < -s \wedge b = 0$	$[t = 0, s = -1, r = 0, b = 0]$	$[2, 1, 1, 2]$

Table 7.4: Non-point curtains generated for 'Hong-90'.

where one notes that the polynomials in y are more numerous as a result due to $x^2 + y^2 - 1$ not being used in (semi-)restricted projection for x. Only $b^2(x-c)^2 + a^2y^2 - a^2b^2$ is used as a pivot for a. There are 5 more polynomials in y on top of the 5 similar ones in usage of multiple equational constraints. The number of non point curtains increases to 409, and the number of point curtains increases to 32. This is unsurprising given the further leaf cells produced via the presence of more univariate polynomials in y, yielding a total of 123665 level 5 leaf cells.

Hong-90

$$\exists a \, \exists b \; r + s + t = 0 \wedge rs + tr + st - a = 0 \wedge rst - b = 0$$

Yet another 5 variable problem, the ordering chosen by ECHeuristic is $[t, s, r, b, a]$.

Multiple Equational Constraints

The full projection bases are:

$$\{s^2 + st + t^2 + a, \mathbf{t^3 + at - b}\}$$
$$\{b - t^3, \mathbf{s^2t + st^2 + b}\}$$
$$\{\mathbf{r + s + t}\}$$
$$\{s, s + t, s^2 + st + t^2\}$$
$$\{t\}$$

and yet again pivot sets are highlighted in bold. Gröbner bases significantly change the input polynomials, which are all ECs, however the size of that Gröbner basis is still 3. There are 9 curtain cells to process after standard lifting, visualised as Table 7.4. None are identified as point curtains in Algorithm 30, which owes to the fact there are packs of neighbouring cells in the cylinder owing to $t = 0$, all sharing the element $b = 0$ in their sample point, which is sufficient to nullify $t^3 + at - b$, regardless of the value of s or r, of which there are several identified by root finding.

B_A, the basis of inequalities from the top level of projection, is $\{s^2 + st + t^2 + a\}$. Full Lazard projection on this generates:

$$\{s^2 + st + t^2 + a\}$$
$$\emptyset$$
$$\emptyset$$
$$\{s^2 + st + t^2\}$$
$$\{t\}$$

And the set of univariate polynomials in t generated entirely coincides with the original set of univariates, $\{t\}$. Hence, there is no extra geometry to merge at the root cell, and we commence with standard lifting on the curtain cells to obtain a total of 771 leaf cells.

Single Equational Constraint

Usage of a single rather than multiple equational constraints changes nothing for this example — the produced projection sets are the same, hence there are the same 771 leaf cells, and the curtains produced the same.

Piano Mover's Problem (Wang)

$\exists a\, \exists b\, \exists c\, \exists d\, a^2{+}b^2 = r^2 \wedge 0 \leq a \wedge b < 0 \wedge 1 \leq c \wedge d < -1 \wedge c{-}(1{+}b)(c{-}a) = 0 \wedge d{-}(1{-}a)(d{-}b) = 0$

Usage of ECHeuristic generates the variable ordering $[b, a, r, d, c]$.

Multiple Equational Constraints

In usage of multiple equational constraints via passing the keyword option `'UseEquations' = 'multiple'`, we encounter a level $3 < n{-}1 = 4$ curtain on the level 4 pivot $ab{-}ad{-}b$ on the cell with sample point $[b = 0, a = 0, r = 1]$, cell description $b = 0 \wedge a = 0 \wedge 0 < r$ and full index $[4, 2, 3]$, and hence are unable to produce a Lazard invariant CAD with multiple equational constraints with this variable ordering.

Single Equational Constraint

Without the danger of production of low level curtains by passing the keyword option `'UseEquations' = 'single'`, we avoid the curtain from the above case, but produce 15 level $n - 1 = 4$ curtain cells. All of them are non point curtains, and are visualised in Table 7.5, where one can see cell indices are such that all the cells have at least one neighbour in the table. B_A in Algorithm 30 is $\{c - 1, bc - cd - b - 1, ad - bc + a + b\}$. Full Lazard projection on this generates the bases:

Description	Sample Point	Cell Index
$b = 0 \wedge a = 0 \wedge 0 < r \wedge 0 < d$	$[b = 0, a = 0, r = 1, d = 1]$	$[4, 2, 3, 5]$
$b = 0 \wedge a = 0 \wedge 0 < r \wedge d = 0$	$[b = 0, a = 0, r = 1, d = 0]$	$[4, 2, 3, 4]$
$b = 0 \wedge a = 0 \wedge 0 < r \wedge -1 < d \wedge d < 0$	$[b = 0, a = 0, r = 1, d = -\frac{1}{2}]$	$[4, 2, 3, 3]$
$b = 0 \wedge a = 0 \wedge 0 < r \wedge d = -1$	$[b = 0, a = 0, r = 1, d = -1]$	$[4, 2, 3, 2]$
$b = 0 \wedge a = 0 \wedge 0 < r \wedge d < -1$	$[b = 0, a = 0, r = 1, d = -2]$	$[4, 2, 3, 1]$
$b = 0 \wedge a = 0 \wedge r = 0 \wedge 0 < d$	$[b = 0, a = 0, r = 0, d = 1]$	$[4, 2, 2, 5]$
$b = 0 \wedge a = 0 \wedge r = 0 \wedge d = 0$	$[b = 0, a = 0, r = 0, d = 0]$	$[4, 2, 2, 4]$
$b = 0 \wedge a = 0 \wedge r = 0 \wedge -1 < d \wedge d < 0$	$[b = 0, a = 0, r = 0, d = -\frac{1}{2}]$	$[4, 2, 2, 3]$
$b = 0 \wedge a = 0 \wedge r = 0 \wedge d = -1$	$[b = 0, a = 0, r = 0, d = -1]$	$[4, 2, 2, 2]$
$b = 0 \wedge a = 0 \wedge r = 0 \wedge d < -1$	$[b = 0, a = 0, r = 0, d = -2]$	$[4, 2, 2, 1]$
$b = 0 \wedge a = 0 \wedge r < 0 \wedge 0 < d$	$[b = 0, a = 0, r = -1, d = 1]$	$[4, 2, 1, 5]$
$b = 0 \wedge a = 0 \wedge r < 0 \wedge d = 0$	$[b = 0, a = 0, r = -1, d = 0]$	$[4, 2, 1, 4]$
$b = 0 \wedge a = 0 \wedge r < 0 \wedge -1 < d \wedge d < 0$	$[b = 0, a = 0, r = -1, d = -\frac{1}{2}]$	$[4, 2, 1, 3]$
$b = 0 \wedge a = 0 \wedge r < 0 \wedge d = -1$	$[b = 0, a = 0, r = -1, d = -1]$	$[4, 2, 1, 2]$
$b = 0 \wedge a = 0 \wedge r < 0 \wedge d < -1$	$[b = 0, a = 0, r = -1, d = -2]$	$[4, 2, 1, 1]$

Table 7.5: The non point curtains generated for 'Piano Movers Problem (Wang)'.

$$\{c - 1, bc - cd - b - 1, ad - bc + a + b\}$$
$$\{b - d, d + 1, ab - ad - b, ad + a + b\}$$
$$\emptyset$$
$$\{a, a - 1, a + b, ab + a - b, ab + a + b\}$$
$$\{b + 2, 2b + 1\}$$

while the original projection bases are:

$$\{c - 1, bc - cd - b - 1, ad - bc + a + b, \mathbf{ab + bc + a}\}$$
$$\{b - d, d + 1, ab - ad - b, ad + a + b\}$$
$$\{a^2 + b^2 - r^2\}$$
$$\{a, a - 1, a^2 + b^2, ab + a - b\}$$
$$\{b, b + 1, b^2 + 1, b^2 + 2b + 2\}$$

with the one equational constraint used as pivot highlighted in bold. In particular $\{b + 2, 2b + 1\} \setminus \{b, b + 1, b^2 + 1, b^2 + 2b + 2\} = \{b + 2, 2b + 1\}$, and hence we obtain the first example where there is a non trivial iteration of Algorithm 31 at level 1. As a result of recursive curtain decomposition, we receive 4, 18, 18, and 140 new cells per level. In total we receive 6501 leaf cells after conclusion of Algorithm 30.

7.3 Case Studies on the Poly-algorithmic Methodology

We investigate case studies on the poly-algorithmic method for examples where it is relevant. We mainly investigate this in terms of the statistics for the CAD(s) used.

Other than the clear varying of the keyword option `'HybridMode'` to control usage of the poly-algorithm, all other options are as default for QuantifierEliminate, most relevant being `'UseEquations'` = `'multiple'`, `'UseGroebner'` = *true*, `'MaxVSDegree'` = 2.

The CAD statistics are deduced from usage of userinfo in Maple — it is sufficient to set `infolevel[ QuantifierEliminate ]` := 2, for example, to deduce such statistics via the associated information printed. The number of leaf cells as a statistic refers to the number of leaf cells upon solution of the last `IQER`.

All examples are from the QE example database unless otherwise stated.

Piano Movers Problem (Wang)

$$\exists a\,\exists b\,\exists c\,\exists d\; a^2+b^2=r^2 \wedge 0\le a \wedge b<0 \wedge 1\le c \wedge d<-1 \wedge c-(1+b)(c-a)=0 \wedge \\ d-(1-a)(d-b)=0$$

The problem is homogeneously existentially quantified. VTS can eliminate 3 out of 4 quantifiers, but the common polynomial $-a^6+a^4r^2+2\,a^5-2\,a^3r^2-2\,a^4+a^2r^2$ appears in all resulting `IQER`s of level 3, which factors into the irreducibles $a^2(a^4-a^2r^2-2\,a^3+2\,ar^2+2\,a^2-r^2)$, of which one is degree 4, and so intraversible for VTS.

The first `IQER` passed to CAD (QEPCADL) forms the QE problem:

$$\exists a\; a^2-r^2\le 0 \wedge a^2-r^2\ne 0 \wedge a\ne 0 \wedge -a\le 0 \wedge a^2-r^2<0 \wedge ((a^3-ar^2-a^2+r^2\le 0 \wedge \\ -a^6+2\,a^4r^2-a^2r^4+2\,a^5-4\,a^3r^2+2\,ar^4-2\,a^4+3\,a^2r^2-r^4\le 0) \vee (a\le 0 \wedge \\ a^6-2\,a^4r^2+a^2r^4-2\,a^5+4\,a^3r^2-2\,ar^4+2\,a^4-3\,a^2r^2+r^4\le 0)) \wedge ((a^2<0 \wedge \\ -a^6+a^4r^2+2\,a^5-2\,a^3r^2-2\,a^4+a^2r^2<0) \vee (-a^2+a\le 0 \wedge (a^2<0 \vee \\ a^6-a^4r^2-2\,a^5+2\,a^3r^2+2\,a^4-a^2r^2<0)))$$

resulting in a CAD using 10 polynomials in projection (5 of which are in r, 5 are in a), and 135 leaf cells on termination (also 154 cells traversed in total). On the next `IQER`, 4 new projection polynomials need to be introduced to extend the CAD. In solving the next 8 `IQER`s no new polynomials are introduced, so the repurposing is purely to accommodate different boolean structures between `IQER`s. The resulting CAD is of 143 leaf cells with 154 total cells traversed in total, and these data are static across these 9 `IQER`s. In the first instance of repurposing, a polynomial is identified as an EC where it was not under the original CAD, but this is purely a semantic identification, because the projection sets are sufficient to describe geometry for the `IQER`s featuring no new polynomials. In other words this EC (a^2-r^2) was already used as an "inequality" polynomial in projection. If QuantifierEliminate were to solve the later `IQER`s first, then this EC would manifest similarly in each, resulting in faster solution of all but what was the first solved `IQER` here.

The case for usage of breadth-wise traversal of the VTS tree to pass over to CAD is the same, considering all `IQER`s formed by VTS are level 1, and the lack of a *true*/*false* quantifier free equivalent means every such `IQER` must be traversed anyway.

Methodology	# Projection Polynomials	# ECs	Total Cells Created	Total Leaf Cells
Depth	10	1	154	143
Breadth	10	1	154	143
Whole	13	0	314	283

Table 7.6: CAD statistics for QuantifierEliminate per poly-algorithmic methodology on 'Piano Mover's (Wang)'.

In contrast, collapsing the whole VTS tree to one QE problem for CAD to traverse results in an existentially quantified disjunction. The structure of this formula is a disjunction of conjunctions due to the elimination of existential quantifiers. Equations are nested in conjunctions amongst other relations that are not equations, so CAD can deduce no equational constraints. The resulting CAD has 13 total polynomials in projection, 7 of which are for r, and 6 for a. There are 283 leaf cells on termination, with 314 created in total including parent cells.

However, usage of this methodology is able to "simplify" away the formula $-r^2 \leq 0 \wedge -r^2 < 0 \wedge -r^2 = 0$ from the output disjunction. This conjunction is clearly equivalent to *false*, but weak simplification on such a formula cannot deduce as such. This comes from a level 1 `IQER` included in the collapsing of the VTS tree. The same is not true when traversing the tree via the poly-algorithmic method, where the `IQER` is ignored.

Collision

$$\exists y\, \exists x\, \exists t\ \frac{(x-t)^2}{4} + (y-10)^2 - 1 = 0 \wedge \frac{(-at+x)^2}{4} + (-at+y)^2 - 1 = 0 \wedge 0 < t \wedge 0 < a$$

Once again the problem is homogeneously existentially quantified, but not fully quantified, with a being a free variable. VTS is able to eliminate $\exists t$ alone, but given x, y, and t all appear quadratically it is not surprising that the degree of the resulting operands increases, and in particular introduces irreducible polynomials in both x and

Methodology	# Projection Polynomials	# ECs	Total Cells	Total Leaf Cells
Depth	65+	1+	?	?
Breadth	65+	1+	?	?
Whole	1328	0	?	?

Table 7.7: CAD statistics for QuantifierEliminate per poly-algorithmic methodology on 'Collision'. "65+" signifies that there are a minimum number of 65 polynomials required, but almost certainly not maximum.

y for every `IQER`. The first such to traverse is

$$\begin{aligned}\exists y\,\exists x\ & a^2 \neq 0 \wedge a^2x^2 - 2\,a^2xy + a^2y^2 - 5\,a^2 \leq 0 \wedge \\ & 125\,a^6x^3 + 500\,a^6xy^2 - 10000\,a^6xy - 75\,a^5x^3 - 300\,a^5x^2y - 100\,a^5xy^2 - 400\,a^5y^3 + \\ & 49500\,a^6x + 2000\,a^5xy + 8000\,a^5y^2 - 5\,a^4x^3 + 160\,a^4x^2y + 220\,a^4xy^2 - 9900\,a^5x \\ & - 39600\,a^5y + 3\,a^3x^3 - 4\,a^3x^2y - 76\,a^3xy^2 - 48\,a^3y^3 + 100\,a^4x - 20\,a^3x - 80\,a^3y \leq 0 \wedge \\ & 25\,a^6x^4 + 200\,a^6x^2y^2 + 400\,a^6y^4 - 4000\,a^6x^2y - 16000\,a^6y^3 - 20\,a^5x^4 - 80\,a^5x^3y \\ & - 80\,a^5x^2y^2 - 320\,a^5xy^3 + 19800\,a^6x^2 + 239200\,a^6y^2 + 1600\,a^5x^2y + 6400\,a^5xy^2 \\ & + 14\,a^4x^4 + 32\,a^4x^3y + 80\,a^4x^2y^2 + 128\,a^4xy^3 + 96\,a^4y^4 - 1584000\,a^6y - 7920\,a^5x^2 \\ & - 31680\,a^5xy + 480\,a^4x^2y - 2560\,a^4xy^2 - 1920\,a^4y^3 - 4\,a^3x^4 - 16\,a^3x^3y - 16\,a^3x^2y^2 \\ & - 64\,a^3xy^3 + 3920400\,a^6 - 2416\,a^4x^2 + 12672\,a^4xy + 9664\,a^4y^2 + a^2x^4 + 8\,a^2x^2y^2 \\ & + 16\,a^2y^4 - 3200\,a^4y + 16\,a^3x^2 + 64\,a^3xy + 15840\,a^4 - 8\,a^2x^2 - 32\,a^2y^2 + 16\,a^2 = 0 \wedge \\ & (-a^3x - 4\,a^3y < 0 \wedge -a^2x^2 - 4\,a^2y^2 + 4\,a^2 < 0 \vee (a^2 \leq 0) \wedge -a^3x - 4\,a^3y < 0 \vee \\ & a^2x^2 + 4\,a^2y^2 - 4\,a^2 < 0) \wedge -a < 0\end{aligned}$$

which is both of high degree (unfortunately for CAD, in the free variable a) and of many operands. The initial CAD for this `IQER` uses one equational constraint, with a total of 65 polynomials in projection, 6 for y, 7 for x, and 52 for a. The problem is unsurprisingly intractable for pure Partial CAD (unless one is to use lifting constraints (Figure 3.4)), even if one is able to observe more progress of the problem via usage of the poly-algorithm.

Usage of the "whole" methodology collapsing the VTS tree results in an initial CAD with 1328 polynomials in projection (1265 for a, 52 for x, 11 for y). There are no equational constraints deduced. The problem is unsurprisingly less tractable due to the far more numerous projection polynomials.

Ellipse A

$$\exists y\,\exists x\ x^2+y^2-1 = 0 \wedge b^2\,(x-c)^2+a^2y^2-a^2b^2 = 0 \wedge 0 < a \wedge a < 1 \wedge 0 < b \wedge b < 1 \wedge 0 \leq c \wedge c < 1$$

A homogeneously existentially quantified problem, but unsurprisingly due to the first two operands, elimination of either x or y results in a formula with an irreducible degree 4 polynomial in the variable not chosen by VTS, due to the doubling in degree

Methodology	# Projection Polynomials	# ECs	Total Cells	Total Leaf Cells
Depth	50	1	10834	7095
Breadth	50	1	10834	7095
Whole	2462	0	?	?

Table 7.8: CAD statistics for QuantifierEliminate per poly-algorithmic methodology on 'Ellipse A'.

resulting from quadratic VTS. Given that there are two quantifiers, all `IQER`s are forced to be level 1. As a result, breadth vs. depth-wise traversal of the tree is trivially the same. The first `IQER` to traverse is:

$$\exists y\ b^2 < 0 \wedge -a^2b^4 + a^2b^2y^2 < 0 \wedge -a^2b^6c + a^2b^4cy^2 - b^6c^3 - b^6cy^2 + b^6c \leq 0 \wedge$$
$$(a^4b^6 - 2\,a^4b^4y^2 + a^4b^2y^4 - 2\,a^2b^6c^2 + 2\,a^2b^6y^2 + 2\,a^2b^4c^2y^2 - 2\,a^2b^4y^4 + b^6c^4 + 2\,b^6c^2y^2$$
$$+ b^6y^4 - 2\,a^2b^6 + 2\,a^2b^4y^2 - 2\,b^6c^2 - 2\,b^6y^2 + b^6 = 0) \wedge -a < 0 \wedge a < 1 \wedge -b < 0 \wedge$$
$$b < 1 \wedge -c \leq 0 \wedge c < 1$$

where one notes the large equation is irreducible and degree 4 in y. Calling QEPCADL on this yields a CAD with 1 equational constraint, 14 polynomials total in b, 13 in c, 9 in a, and 4 in y, making 40 total polynomials in projection and 23 leaf cells. The CAD is then always repurposed amongst the remaining 5 ineligible `IQER`s, where twice no polynomials need to be added, and otherwise 10 polynomials are added cumulatively. In total 50 polynomials are used, and on termination the CAD is of 5071 leaf cells. The same EC is able to be identified and coerced for each `IQER`.

Using standard methodology to perform Partial CAD on the formula formed by the whole VTS tree leads to a cumulative projection basis consisting of 2462 polynomials with no identification of ECs, which is intractable to lift within the allotted time span.

One notes that the `IQER` above is in principle amenable to usage of lifting constraints via the latter operands defining a "box" being untouched by action of VTS. The poly-algorithm does not currently attempt to use lifting constraints, because there is no assumption later `IQER`s could be consistent with last usages of lifting constraints, which makes repurposing of the CAD tree difficult.

Off-Center Ellipse

$$a \neq 0 \wedge \forall x\, \forall y\ 16\,a^2y^2 - 8\,a^2y - 3\,a^2 + 4\,x^2 - 4\,x + 1 = 0 \Rightarrow x^2 + y^2 \leq 1$$

After conversion to prenex formula we obtain the universally quantified $\forall x \forall y\ a \neq 0 \wedge 16\,a^2y^2 - 8\,a^2y - 3\,a^2 + 4\,x^2 - 4\,x + 1 \neq 0 \vee x^2 + y^2 \leq 1$. Unlike the most canonical case for a universally quantified formula, the outer operator is a conjunction after this conversion. Despite that, in proceeding with poly-algorithmic QE, our first `IQER` to

Methodology	# Projection Polynomials	# ECs	Total Cells	Total Leaf Cells
Depth	20	0	749	735
Breadth	20	0	749	735
Whole	28	0	1581	649

Table 7.9: CAD statistics for QuantifierEliminate per poly-algorithmic methodology on 'Off-Center Ellipse'.

examine is

$$\begin{aligned}\forall x\ a^2 \neq 0 \vee a^2 = 0 \vee (a \neq 0 \wedge 9\,a^4 - 24\,a^2x^2 + 16\,x^4 + 24\,a^2x - 32\,x^3 - 6\,a^2 + 24\,x^2 \\ - 8\,x + 1 \neq 0 \vee\ 64\,a^4x^2 - 55\,a^4 - 24\,a^2x^2 + 16\,x^4 + 24\,a^2x \\ - 32\,x^3 - 6\,a^2 + 24\,x^2 - 8\,x \leq -1)\end{aligned}$$

which yields a CAD with 6 polynomials in a, 2 in x, 8 total without any equational constraints. 15 leaf cells are required to solve this `IQER`. Unlike the case for the prenex top level formula, the `IQER`'s formula is an outer disjunction, as is the minimal example we'd like to act upon for a universal quantifier. As the maximal level for an ineligible `IQER` is 1, all ineligible `IQER`s are level 1, and depth-wise vs. breadth-wise traversal is identical. The next `IQER` introduces 8 polynomials in a, 4 in x, which yields us our maximal projection sets of 20 polynomials, and the last `IQER` to examine requires 739 leaf cells. This is in principle larger than the CAD used to solve the "whole" QE problem formed by VTS, requiring just 649 leaf cells, but more cells in total pass through CCHILD to yield these (1581 as opposed to 749), and there are 8 more polynomials in total necessary beforehand. The raw benchmarking data produced for Section 7.4.4 implies that despite the more numerous leaf cells from the poly-algorithmic approach, the methodology is still faster, albeit producing a longer formula.

Joukowsky Upper Half Plane

$$\begin{aligned}\forall a\,\forall b\,\forall c\,\forall d\ a(c^2+d^2)(a^2+b^2+1) - c(a^2+b^2)(c^2+d^2+1) = 0 \\ \wedge\, b(c^2+d^2)(a^2+b^2-1) - d(a^2+b^2)(c^2+d^2-1) = 0 \wedge 0 < b \wedge 0 < d \Rightarrow \\ a - c = 0 \wedge b - d = 0\end{aligned}$$

After conversion to prenex form, we obtain

$$\begin{aligned}\forall a \forall b \forall c \forall d\ 0 \neq a^3c^2 + a^3d^2 - a^2c^3 - a^2cd^2 + ab^2c^2 + ab^2d^2 - b^2c^3 - b^2cd^2 - a^2c + ac^2 + ad^2 \\ \vee\ - b^2c0 \neq a^2bc^2 + a^2bd^2 - a^2c^2d - a^2d^3 + b^3c^2 + b^3d^2 - b^2c^2d - b^2d^3 + a^2d + b^2d - bc^2 \\ -bd^2 \vee 0 \leq -b \vee\ 0 \leq -d \vee\ a - c = 0 \wedge b - d = 0,\end{aligned}$$

a fully homogeneously universally quantified formula, and in particular first fully quantified formula amongst the case studies. It takes a long time to yield the ineligible `IQER`s

in the first place. For the first time, we receive IQERs of non uniform level, hence depth-wise and breadth-wise traversal act differently for the first and only time amongst the case studies. An ineligible IQER of greatest depth is

$$\forall a\ a^2 \neq 0 \vee\ a \leq 0 \vee\ -a^4 \leq 0 \wedge a^4 \neq 0 \vee\ a^2 \leq 0 \vee -4\,a^{11} - 3\,a^9 \leq 0 \wedge 4\,a^{11} + 3\,a^9 \neq 0$$
$$\vee\ 8\,a^{11} + 10\,a^9 + a^7 \leq 0 \wedge 8\,a^{11} + 10\,a^9 + a^7 \neq 0 \vee\ -400\,a^9 - 100\,a^7 - a^5 \leq 0 \wedge$$
$$400\,a^9 + 100\,a^7 + a^5 \neq 0 \vee\ 36\,a^7 + a^5 \leq 0 \wedge 36\,a^7 + a^5 \neq 0 \vee\ -a^5 < 0 \vee\ -a^3 \leq 0$$
$$\wedge a^3 \neq 0 \vee\ a \leq 0 \vee\ a = 0 \wedge a^3 - a = 0 \wedge a^3 \leq 0 \wedge a^3 \neq 0 \vee\ -a \leq 0 \wedge a^3 = 0$$
$$\wedge a^5 - a = 0.$$

while an ineligible IQER of lesser depth is

$$\forall a \forall b\ a^2 + b^2 \leq 0 \vee a^3 + ab^2 + a \leq 0 \vee -a^{12} - 4\,a^{10}b^2 - 6\,a^8b^4 - 4\,a^6b^6 - a^4b^8 + 4\,a^8b^2 +$$
$$12\,a^6b^4 + 12\,a^4b^6 + 4\,a^2b^8 + 2\,a^8 + 12\,a^6b^2 + 18\,a^4b^4 + 8\,a^2b^6 + 4\,a^4b^2 + 4\,a^2b^4 - a^4 \leq 0 \vee$$
$$a^{19}b^2 + 7\,a^{17}b^4 + 21\,a^{15}b^6 + 35\,a^{13}b^8 + 35\,a^{11}b^{10} + 21\,a^9b^{12} + 7\,a^7b^{14} + a^5b^{16} + 3\,a^{17}b^2 +$$
$$18\,a^{15}b^4 + 45\,a^{13}b^6 + 60\,a^{11}b^8 + 45\,a^9b^{10} + 18\,a^7b^{12} + 3\,a^5b^{14} + a^{15}b^2 + 5\,a^{13}b^4 +$$
$$10\,a^{11}b^6 + 10\,a^9b^8 + 5\,a^7b^{10} + a^5b^{12} - 5\,a^{13}b^2 - 20\,a^{11}b^4 - 30\,a^9b^6 - 20\,a^7b^8 - 5\,a^5b^{10}$$
$$- 5\,a^{11}b^2 - 15\,a^9b^4 - 15\,a^7b^6 - 5\,a^5b^8 + a^9b^2 + 2\,a^7b^4 + a^5b^6 + 3\,a^7b^2 + 3\,a^5b^4$$
$$+ a^5b^2 \neq 0 \vee b \leq 0 \vee -a^9 - 3\,a^7b^2 - 3\,a^5b^4 - a^3b^6 + a^7 + 6\,a^5b^2 + 9\,a^3b^4 + 4\,ab^6 + a^5$$
$$+ 5\,a^3b^2 + 4\,ab^4 - a^3 \leq 0 \vee a^3 + ab^2 - a = 0 \wedge -a^3b - ab^3 - ab \leq 0 \wedge a^9 + 7\,a^7b^2$$
$$+ 15\,a^5b^4 + 13\,a^3b^6 + 4\,ab^8 - a^7 - 2\,a^5b^2 - a^3b^4 - a^5 - 5\,a^3b^2 - 4\,ab^4 + a^3 = 0.$$

The quantifier free equivalent of the original formula is *true*, so as it happens QE necessarily needs to traverse every IQER in some sense to deduce a quantifier free equivalent, because of no meaningful truth value. To trace depth-wise traversal of the tree, our starting IQER yields 8 polynomials in a, so 8 total polynomials, and 7 leaf cells are sufficient. 42 polynomials in a are added in traversing all the maximum level IQERs, with the CAD being reused each time, and quite often the CAD is sufficient in the sense no polynomials need to be added. In reaching the first level 2 IQER in a and b, we need to add 4 and 7 polynomials in a and b respectively, and the CAD extends canonically due to the fact the new variable is quantified similarly to a. Lifting failures related to evaluation of truth values occur in further IQERs, which are unrecoverable because QE categorically know the quantifier free equivalent of such in lieu of a meaningful truth value anywhere else. QuantifierEliminate is finding the CAD increasingly intractable, failing to evaluate various IQERs in search for a meaningful truth value before timeout. Any which way, the CAD is always reused amongst these IQERs, and the total number of polynomials in projection bases before timeout is 123. In breadth-wise traversal, we actually begin with an IQER of level 1 — quantified with a, b and c. The IQER yields 944 polynomials in projection, so QEPCADL does not make the timeout, but in presence of the IQERs of other levels, we know that this is likely not the minimal set of projection polynomials required.

For usage of CAD in a "whole" sense, we cannot even complete one step of projection due to the enumerable and large polynomials created in usage of VTS, so are uncertain how many polynomials would be yielded in the projection bases.

Methodology	# Projection Polynomials	# ECs	Total Cells	Total Leaf Cells
Depth	123+	?	?	?
Breadth	944+	?	?	?
Whole	?	?	?	?

Table 7.10: CAD statistics for QuantifierEliminate per poly-algorithmic methodology on 'Joukowsky Upper Half Plane'. "+" signifies that at least as many that polynomials are required in projection with these methodologies.

7.3.1 Conclusions

One notes that the VTS trees traversed by the poly-algorithm in every case are almost always alike in terms of the fact depth and breadth wise traversal are entirely alike due to the uniform levels of the ineligible `IQER`s produced. Additionally, the CAD used is *always* repurposed, i.e. the "poly-share ratio" for every `IQER` on the retained CAD always exceeds `POLY_SHARE_THRESHOLD`$= \frac{1}{2}$ for every `IQER`. Usage of the whole methodology often produces more simplified output due to CAD "being its own simplifier", so where the CAD is omitted from usage on individual `IQER`s, we fail to provide any simplification (such as with the "Piano Mover's" case study), or when attributing CAD results to individual `IQER`s, this may lead to some redundancy amongst output formula operands.

Only one example is fully quantified, and its quantifier free equivalent is not a meaningful truth value for the quantifier — instead the opposite. This same example does however provide cause for extension of the CAD to a new quantifier when depth-wise traversal is used. A non trivial example that is fully homogeneously quantified, and yields a meaningful truth value is created out of an existing example in Figure 4-2 to exemplify poly-algorithmic witnesses.

Comparisons of usage of poly-algorithmic QE against QE by pure Partial CAD are largely left to Section 7.4.4, and meanwhile one can also see that the case studies leading to smaller CADs with the poly-algorithmic methodology also lead to better performance in these benchmarks, sometimes allowing examples to finish where they otherwise would not without such methodology. Further case studies would be useful to investigate cases where the "whole" methodology outperforms non trivial usage of the poly-algorithm, or examples where the poly-share ratio is not met for successive `IQER`s.

7.4 Benchmarking

7.4.1 Methodology of Benchmarking

[6] discusses the typical methodology of benchmarking SAT & SMT software, imploring the wider symbolic computation community to use the same methodology. In particular, it introduces survival/cactus plots to format the data in an intelligible way where usage of a timeout threshold is necessary. We cite the definition of a survival plot as follows:

1. For each method separately:
   (a) Solve each problem p_i, noting the time t_i (up to some threshold T).
   (b) Sort the t_i into increasing order (discarding the time-out ones).
   (c) Plot the points $(t_1, 1)$, $(t_1 + t_2, 2)$ etc., and in general $(\sum_{i=1}^{k} t_i, k)$.
2. Place all the plots on the same axes, optionally using a logarithmic scale for time.

N.B. There is therefore no guarantee that the same problems were used to produce time results from different solvers.

Survival plots are effective in the sense they encapsulate a lot of information in a very concise way — one can see how many benchmarks finish in total and the cumulative time taken, and one can aggregate data with respect to other varying "dimensions" of the data while still being able to examine another single dimension of the data effectively. For this project, each "problem" is a QE problem or an unquantified formula for CAD, and the methods are various implementations of QE and/or CAD, or implementations of variable strategy for CAD and generation of the associated projection bases. The benchmarking here always uses a logarithmic scale for time.

All benchmarking was undertaken on a computer running Ubuntu 18.04.3 with 16GB of RAM and an Intel i5-4590T CPU running at 2.00 GHz. Where Maple is concerned, the version is Maple 2020.1. Care must be taken to ensure any effects of Maple caching or any other properties of a Maple session are avoided (one recalls that polynomial operations in projection in `QuantifierElimination` are always cached). Hence, all benchmarks for any packages implemented in Maple are run in their own Maple session. The timing and memory data are provided by wrapping relevant function calls with CodeTools:-Usage, and providing the argument `'output'` = [cputime, bytesused, output] to obtain the amount of time and memory used while also obtaining the output. This also avoids including Maple startup time into the benchmarks. Timeouts are achieved by calls from the operating system (here Linux Ubuntu using timeout). This is for two reasons: Maple's timelimit is unreliable at ceasing low level `C` code in subroutines, and because Maple's timelimit is inapplicable for software implemented outside Maple. Because every benchmark needs its own Maple session, the iteration over different packages, examples, and/or options has to be abstracted to something that can call Maple, i.e. scripting with `bash`. Hence any benchmarking process here consists of usage of Maple scripts that takes "arguments" from `bash` via a sequence of startup commands to Maple via "`-c`". These arguments define what example to benchmark on, the function to use, and any other data relevant to writing the benchmarking data, including the path to the "comma separated values" (`.csv`) file to write data to. Any row of such a `.csv` corresponds to data for exactly one benchmark. Via the Maple option `-c`, we can define variables for Maple intelligible to the Maple script to run to coerce the benchmarking process to benchmark what is required. Usually the initial data such as the name of the example, function to be used, etc. is written by Maple as the first portion of a new line of the `.csv`, and if Maple completes the example before the time limit, writes the rest of the line of the `.csv`, including the time

taken, etc. Otherwise, on time outs, `bash` detects that Maple has timed out via the prescribed exit code for time outs, 124, and writes the rest of the line to notify that the benchmark timed out. Maple writes data via fprintf, while `bash` writes data by printf. Other options passed to Maple are ones such as `-q`, and `-b` to define the list of archives to read from to import e.g. `QuantifierElimination` and `RegularChains` (for parity, any Maple package is always read from `.mla`). The timeouts to use vary between the sections, because some benchmarks to perform are more numerous than others. The Maple scripts also catch any relevant errors from functions to benchmark such that that information can be written when necessary (often, the errors are expected in some capacity, i.e. lifting failures). In writing comma separated value files, the data is easily readable by software such as Excel, MATLAB, or even Maple for processing — the survival plots are generated via Excel after sorting relevant rows of data.

Benchmarking against `QEPCAD B` presents a couple challenges via the fact that it is interactive software. This means that the example to benchmark must be redirected in, and the output redirected to a file. Timings for `QEPCAD B` are via its own output, where this is gleaned from the output that `QEPCAD B` itself writes. `QEPCAD B` also includes no intrinsic variable orderings, so it is only meaningful to benchmark where an ordering is provided.

The semantics of `.csv` files produced are that `Inf` in a timing or memory column corresponds to the software running out of time (i.e. exceeding the time limit set) or running out of memory (deduced by Maple producing an error about failing to allocate enough memory for the computation, which can be caught). In this case the other respective column will contain -1. -1 in both columns refers to the case where an error occurred. Such an error can be:

- An expected error in a mathematical sense — a curtain or nullification occurrence in the sense of Section 3.7.2,
- An unexpected error in a mathematical sense — any of the other lifting errors discussed specifically in Section 3.7.2, usually specifically under `QuantifierElimination`,
- Or any other bug in very low level code, including that of Maple's, of which each implementation usually features at least one.

One notes the "mathematical errors" from above are often induced in the sense we request `ProjectionCAD` to error out on such, or we use multiple equational constraints in `QuantifierElimination`'s CAD despite its incompleteness in terms of low level curtains. In this case it is obvious that we are timing completion of CAD or QE only on mathematically correct output. Failures of `QEPCAD B` to finish are less apparent as the scripts do not attempt to deduce why `QEPCAD B` failed to finish in detail beyond when it is apparent via exceeding a time limit, but any other error code not associated with a time out is printed to the right of the "Error" column.

7.4.2 CAD Variable Strategies

The benchmarking here examines the time taken to produce the variable ordering & projection to use per example in the CAD and QE databases. The main reason for amalgamating this investigation is that for three of the strategies implemented by `QuantifierElimination`, generation of all projection bases is intrinsically tied to generation of the ordering (Section 3.8). Usage of any of these is worst case $\mathcal{O}(n!)$ in practice due to commutativity of similarly quantified or unquantified variables, but here we treat all examples as unquantified to enable full scope for variable strategy, to reconcile with the benchmarking process in Section 7.4.3, where the variable orderings were generated independently, untimed, for each strategy, before generating the full CAD via each implementation using that ordering. This goes some way to investigate which orderings yield CADs with the fewest possible cells, and one can also begin to separate the cost of projection from lifting (where projection and lifting is applicable, i.e. not `RegularChains`). Lifting follows from projection, and the cost of lifting is intrinsically tied to that of projection. Although not a variable strategy supplied by `QuantifierElimination`, we also make use of the variable strategy for CAD implemented by `RegularChains`, SuggestVariableOrder, which takes one set of polynomials. Here, we supply $A \cup E$ for the canonical sets A and E supplied to projection in `QuantifierElimination`, and the keyword optional argument `'decomposition' = 'cad'` to indicate that we are interested in a CAD relevant variable ordering. In particular, `RegularChains`' variable ordering does not distinguish ECs as part of its variable strategy. SuggestVariableOrder is functionally a heuristic, in the same manner as Brown or ECHeuristic, in that it only attempts to examine trivial data. Usage of SuggestVariableOrder before CAD in `QuantifierElimination` could be achieved by a user by usage of `QuantifierTools`' GetAllPolynomials with `'output' = 'list'` on the desired formula, which produces a list amenable to passing to SuggestVariableOrder in this way. The full list of variable strategies to benchmark, essentially all discussed previously in Section 3.8, is as follows:

- ndrr (Definition 67)
- sotd (Definition 66)
- ECHeuristic (Algorithm 36)
- Brown ([13], "Brown heuristic")
- Greedy ([25], Section 3.8)
- `RegularChains`' SuggestVariableOrdering with usage of the keyword option `'decomposition' = 'cad'`.

Although it is not documented as such within Maple help pages or otherwise, RegularChains:-SuggestVariableOrdering actually implements the Brown heuristic on usage of the keyword option `'decomposition' = 'cad'`. The variable strategy offered by this function was originally incorporated in the benchmarking of this work assumed

to implement a distinct heuristic from one offered by `QuantifierElimination`, including one that could cater specifically to a CAD under the differing methodology that `RegularChains` offers to construct a CAD.

These benchmarks are generated with every single combination of the keyword options relevant to projection in `QuantifierElimination`'s CAD, `'UseEquations'`, `'PropagateECs'`, `'UseGroebner'`, and because of the combinatorics of such data, Figure 7-1 aggregates all data generated to just examine the dimension of usage of strategy as a survival plot.

The file `var_order_testing.sh` is the `bash` script iterating across the CAD and QE example sets, starting fresh sessions of Maple to run the Maple script `var_order_benchmark.mpl` with "arguments" to define the variable strategy, example, and other options to use for projection. Together, they write the file `ProjectionBenchmark.csv` found in the repository [65]. The Maple script is essentially benchmarking the `QuantifierElimination` function CADChooseVarsProjection, which handles variable strategy and projection according to the value of the keyword option `'VariableStrategy'` passed, which can be a list of variables if generated already (here, via usage of the "heuristics"), else a symbol defining a strategy. Of course CADChooseVarsProjection must also take into account the keyword options corresponding to usage of equational constraints. Additionally, generation of the file `HeuristicBenchmark.csv` by the relevant scripts for variable strategy benchmarking verifies that usage of the heuristics in themselves (without projection) is completely negligible for each, hence why the orderings given by the heuristics are generated before passing to CADChooseVarsProjection, but the end result is the same as having passed e.g. `'VariableStrategy' = 'Brown'` in this case, because projection immediately follows from the generated ordering. Figure 7-1 is a survival plot of the timings for generation of ordering & projection per strategies as discussed, where the plot is generated from `ProjectionBenchmark.csv`. The benchmarking data including the `.csv`s was generated by usage of the `bash` script `var_order_testing.sh`.

The use of the greedy strategy appears to perform best here. Usage of greedy is less expensive than that of "full sotd" while still generating $\mathcal{O}(n)$ projection sets for each choice of x_1 in this case. Considering `QuantifierElimination`'s attention to equational constraints corresponding to the keyword options supplied in any context, variable strategies such as greedy generating and examining the projection sets will be expectedly sensitive to usage of equational constraints (Remark 68). However, one must also take into consideration that `QuantifierElimination` finds usage of Gröbner basis preprocessing to be incompatible with usage of the greedy CAD variable strategy (Section 3.8)), so the timings never include generation of a Gröbner basis where equational constraints are concerned, which may flatter the timings for *projection* for greedy, but we may not see the benefits in terms of number of leaf cells produced in the benchmarking of the next subsection, where we note one of the intentions of usage of Gröbner bases via the methodology implemented in `QuantifierElimination` is to remove spurious roots arising from propagation of equational constraints.

ECHeuristic is essentially equivalent to usage of the Brown heuristic for most examples, which is unsurprising considering not every example features ECs, and ECHeuris-

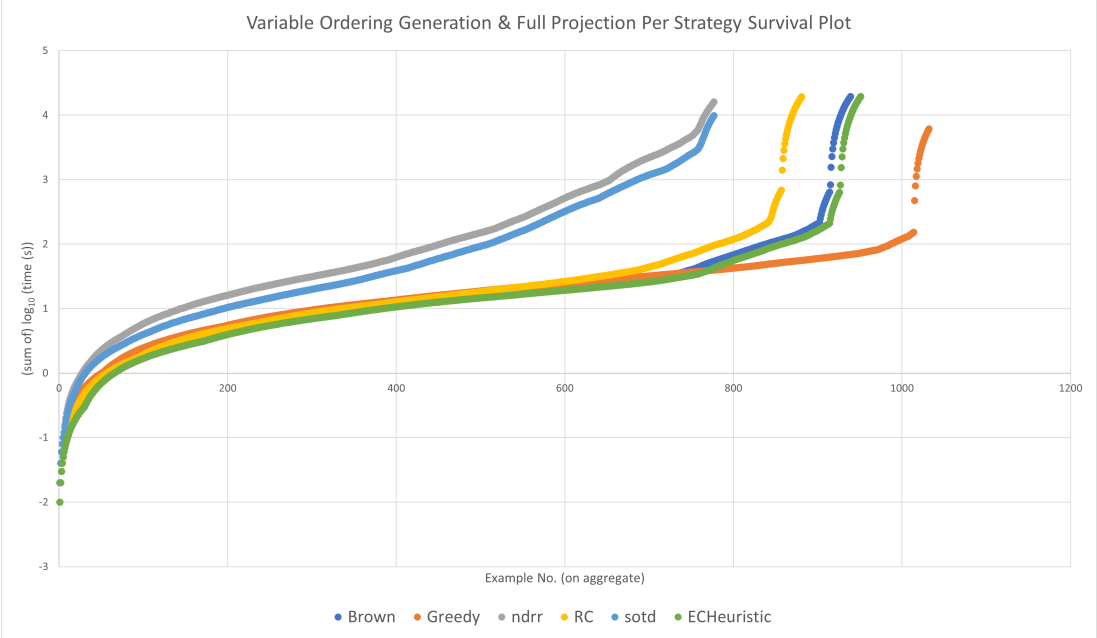


Figure 7-1: Survival plot for usage of each variable strategy for CAD implemented in `QuantifierElimination`, and that defined by RegularChains:-SuggestVariableOrder against time taken to generate the variable ordering & projection (in seconds, with a logarithmic scale). The data points are on aggregate in terms of the other options used in terms of ECs (propagation of equational constraints, Gröbner bases, and usage of anything up to multiple ECs).

tic is equivalent to usage of Brown in this case. Overall, we see a slight improvement from ECHeuristic via the tail end of the curve, most likely as a result of the extra attention to equational constraints. The shape of the curve for `RegularChains`' heuristic is similar to that of Brown or ECHeuristic, and considering in hindsight `RegularChains`' heuristic actually implements Brown's heuristic, one must consider why it does not entirely imitate `QuantifierElimination`'s Brown heuristic. One reason could be that `RegularChains` pays attention to e.g. partial factorisations offered by formulae when computing degrees, or that it makes different choices in ordering variables under genuine ties after all three tiebreakers of the Brown heuristic. Such a choice would be genuinely arbitrary. That being said, the relative significance of the difference in the curve between `RegularChains`' heuristic with `QuantifierElimination`'s Brown implies the gain by ECHeuristic suggests that the implied gain of ECHeuristic may not be largely significant.

To conclude inspection of the strategies, one sees that ndrr and sotd live up to their $\mathcal{O}(n!)$ expectations. ndrr carries the expected overhead of real root isolation on the univariate bases generated amongst all $n!$ `projection` objects. This root isolation must support real root isolation of polynomials with coefficients that are real algebraic numbers, i.e. the implementation uses `QuantifierElimination`'s wrapper isolateRootsOf (Section 3.4.1). Input polynomials from examples involving radicals are converted to

real algebraic numbers in terms of RootOfs on parsing for `QuantifierElimination`, and such RootOfs would be carried throughout the full projection process such that the univariate projection bases could feature irrational coefficients. However one notes that very few of the examples in the databases used here feature radicals. Hence the concerns raised by Section 3.4.1 on root isolation of polynomials over irrational real algebraic numbers are likely not of large consequence for ndrr. One notes that usage of ndrr and sotd may compute $\mathcal{O}(n!)$ different Gröbner bases for sets of ECs per each ordering to "test", which is of course included in the timings.

Additionally, one notes that for the the sotd and ndrr variable strategies, that `QuantifierElimination` caches the results of expensive polynomial operations in projection (Section 5.2.1). The purpose of this is to support incremental projection via caching, but this caching is always enabled. Hence, polynomial operations called from ordering to ordering may coincide to some extent. For example, if two orderings amongst those valid permutations such that x_n and x_{n-1} are the same are tested in succession, we have retained the results of projection on x_n via caching, which is a win. To make this process even faster considering the caching, it would be best to ensure that this is coerced, such that one iterates across valid permutations such that the last variable is fixed for as long as possible, to ensure that we retain the projection bases for x_n in the cache each time. The current implementations of sotd and ndrr as variable strategies unfortunately have the opposite behaviour in usage of Maple's combinat:-permute to generate the variable ordering permutations, which instead in effect fixes x_1 for as long as possible in the generated list — this list would require sorting in order to have the desired behaviour in total. A current loss here is that because Maple's implementation of caching (via `option cache`), is a "Least Recently Used" (LRU) implementation (that forgets the results of arguments least recently used due to the necessary maximum cache size), some of the results of projection may be "forgotten" from the cache by the time of the first usage of incremental projection when sotd or ndrr are used, if the best possible ordering in terms of those metrics was found very early on amongst those permutations to iterate across. For `QuantifierElimination`, the maximum cache size for discriminants is 512, and twice that for resultants (due to the enforced symmetry).

Largely, the variable strategies fall into three classes — the greedy strategy, usage of the pure "heuristics", and usage of the $\mathcal{O}(n!)$ strategies, but projection and variable orderings are a means to an end, and so this analysis merely prefaces their performance in usage to generate a full CAD including the lifting. That being said, timeouts where projection is concerned preclude the option of usage of the projection to generate the lifting, which is reflected in the benchmarking methodology of the following subsection.

7.4.3 Cylindrical Algebraic Decomposition

We benchmark various implementations of CAD in Maple with an interest in CADs for unquantified formulae or sets of polynomials to enable full scope for variable strategy, while taking into account aspects of formula structure (in particular equational constraints). A description of the background of the other implementations can be found in Section 1.3. In some sense the benchmarking of this section imitates previous

investigations such as that of [38] in benchmarking sotd, ndrr and Brown, but includes further strategies, and takes a specific view to the Lazard projection with ECs where appropriate.

- `QuantifierElimination`'s CylindricalAlgebraicDecompose or PartialCylindricalAlgebraicDecompose depending on if the example attributes boolean structure or not.
- `ProjectionCAD`, including e.g. ECCAD, TTICAD, which is a CAD with respect to designated equational constraints or a truth table invariant CAD for a formula respectively. Meanwhile CADFull provides a full CAD for examples attributing no boolean structure, i.e. lists of polynomials.
- `RegularChains`' SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose, which takes a list of polynomials (for the "full CAD" case), else a "list of semi-algebraic systems", where the latter is essentially equivalent to a formula in disjunctive normal form, hence attributing boolean structure, via a list of lists, where the inner lists are conjunctions of relations.

`QuantifierElimination` includes standalone implementations of CAD via CylindricalAlgebraicDecompose and PartialCylindricalAlgebraicDecompose, the former of which is a tool to produce full CADs for formulae or sets of polynomials and the latter QE via full CAD. Although CylindricalAlgebraicDecompose accepts (quantified) Real Tarski formulae, because of the intention to produce every single maximum leaf cell, it never evaluates truth values on the leaf cells it yields. It essentially decomposes the formula as the canonical sets A and E and performs full CAD. It can also accept pure sets of polynomials without relational operators, in which case evaluation of truth values is ill defined. The benchmarking on CAD here uses the CAD and QE databases described at the beginning of this section. Usage of the economics examples would not generate very useful or interesting data, with most examples being purely linear in many variables, which is not a hugely interesting case for projection or lifting in CAD.

The pure CAD examples are rather inhomogeneous in type, some being lists of polynomials, or relations, else just unquantified formulae. Because most examples are formulae, they attribute structure, including equational constraints. We want to allow the benchmarked software to identify any boolean structure that it can possibly use. Additionally, `RegularChains` has the semantics of only returning cells holding the truth value *true* on formulae (lists of semi algebraic systems). `RegularChains` is used with default option arguments, in particular those enabling it to use its incremental methodology and enabling it to identify and use optimizations related to boolean structure and ECs in its own way (`'method' = 'recursive'` and `'optimization' = 'TTICAD'`). Lastly `'output' = 'cadcell'` is specified as an option such that we can easily enumerate the number of cells returned by `RegularChains`. PartialCylindricalAlgebraicDecompose accepts an unquantified formula, and in this case produces a `CADData` object, which can be examined to enumerate statistics on cells, such as the number of cells with truth value *true*. In this way Partial CAD is extended to the case where $m = 0$. In this case there is no scope for removal of CAD subtrees via propagation of truth values, however CAD subtrees may truncate at cells with a determinate

truth values with respect to the unquantified input formula, and so we may not construct every single level n cell, in contrast to the case for full CAD. When an example is not a formula, but e.g. a list of polynomials, this is passed to a different appropriate function or with different input semantics such that a full CAD is generated instead. Because `RegularChains` only returns a list of cells with an implicit truth value of *true*, and `ProjectionCAD` returns all leaf cells even when passed something attributing boolean structure, the number of cells generated is not directly comparable between these two packages. This is because `ProjectionCAD`'s input understands and requires some boolean structure, such as the designated ECs, but cannot evaluate truth values because it only understands that certain polynomials owe to "inequalities", without a specific relational operator in this case. However, two columns in the `.csv` generated appear, one for enumeration of the total number of leaf cells, and the other for number of *true* cells. `QuantifierElimination CADData` objects are amenable to examination of both statistics, by usage of the method NumberOfLeafCells, with keyword option `'TruthValue'` = *true* in the latter case. Therefore there always exists an appropriate cell statistic from `QuantifierElimination` comparable to either `RegularChains` or `ProjectionCAD`. To convert formulae to a "list of semi algebraic systems" as is required for `RegularChains`, we can use the not directly exported function TRDcadLogicFormulaToLsas from that module. `QEPCAD B` is only relevant where quantifier elimination is concerned, so does nothing non trivial on the unquantified examples to use here and is not benchmarked. One notes that failure to evaluate the truth value of a cell is one type of lifting failure associated with `QuantifierElimination` where convoluted RootOfs are produced. In the fully unquantified case as such as this, lifting failure recovery is not particularly relevant due to lack of removal of CAD subtrees via propagation of truth values, so we note that unfortunately `QuantifierElimination` may fail to complete various benchmarks due to (precedented) errors. Because CylindricalAlgebraicDecompose does not evaluate truth values, this type of lifting failure is not relevant when CylindricalAlgebraicDecompose is used, but we have interest in generating and comparing the same functionality as `RegularChains` on unquantified formulae. Again, this type of lifting failure is due to the developmental usage of the particular representation of real algebraic numbers.

Other lifting failures can be prevalent in any case, including curtains, such as low level curtains in varying usage of `'UseEquations'` for `QuantifierElimination`. We force production of errors from `ProjectionCAD` via the keyword option `'failure'` = `'err'` such that we can examine the frequency of nullification occurrences, in particular in comparison with the frequency of Lazard curtains from `QuantifierElimination`. `RegularChains` has no such analogous type of error to inspect. Examples that are principally Real Tarski formulae featuring radicals fail for `RegularChains` and `ProjectionCAD`, being invalid input for those packages.

Any one invocation of timelimit uses a timeout of 750 seconds for this benchmarking, to match the timeout used for the projection benchmarks of Section 7.4.2. In this section, all examples are treated as unquantified, to give full scope to variable strategy, exaggerating the effects of the generated variable orderings (in practice in the context of QE, only similarly quantified or unquantified variables commute). Primarily, the variable orderings to use are those generated from the strategies used in the previous

section 7.4.2. This is of course to extend upon the investigation of that section for `QuantifierElimination`, to examine the efficacy of the lifted CADs having examined the performance of generation of the full projection sets with ordering. For what it's worth, we also obtain information as to whether usage of these strategies are ever effective for other implementations of CAD, especially `RegularChains`, which is not a projection & lifting CAD where `ProjectionCAD` and `QuantifierElimination` are. We note that some of the projection based strategies generate orderings informed by information specific to `QuantifierElimination`, such as usage of the Lazard projection with equational constraints, which does not immediately reconcile with the other packages. Therefore it may be unsurprising that the orderings generated here cater only to `QuantifierElimination`, but we still obtain data comparing the strategies in terms of `QuantifierElimination` at the very least.

Hence we generate every ordering independently in a separate Maple session using CADChooseVarsProjection in a similar manner to that of the previous section, which is the job of the Maple script `cad_var_order.mpl`, taking into account the passed "argument" of a `QuantifierElimination` variable strategy to use. CADChooseVarsProjection is the `QuantifierElimination` function generating variable ordering via strategy and generating the full projection sets, so in this call being time limited by timeout, only the same example and variable strategy pairings not timed out in terms of the previous projection benchmarking manage to be reused here. The variable ordering is written to a file and picked up by Maple with the script `cad_benchmark.mpl`, which actually performs the CAD benchmark for whatever example and package, preparing the appropriate arguments to pass to an appropriate function to benchmark. The variable strategies to use are the same as those from the previous subsection. Generation of the variable orderings is also time limited by timelimit, because of the number of examples to process. If generation of the variable ordering times out for an example, clearly the benchmark cannot be performed for any of the packages. The iteration over examples and packages is handled by `bash` as usual, with the shell script `cad_testing.sh`. One notes that several of the examples time out in terms of projection in the previous subsection. The options used in generation of the variable orderings only vary in terms of usage of number of equational constraints used (`'none'`, `'single'`, or `'multiple'`). For whatever options are used in generation of the variable ordering, `QuantifierElimination` follows suit in usage of those options for the CAD benchmark. The data recorded alongside a benchmark corresponding to the options used in generation of the variable ordering is unintelligible beyond generation of the variable ordering for `ProjectionCAD` and `RegularChains`, where such options related to equational constraints would be irrelevant.

In addition, and perhaps most pertinently, the last set of benchmarks to perform aside from the above is by running each package on each example with usage of its "own" variable strategy, that is it gets to define variable strategy in whatever way is intended as default for those packages. In addition, the benchmarking also varies usage of equational constraints and Gröbner bases in terms of the keyword options `'UseEquations'` and `UseGroebner` for `QuantifierElimination` in this case. In the case of `ProjectionCAD`, the default variable strategy offered involves weighted usage of the ndrr and sotd metrics on projection bases to decide, so it is relevant to note that

here timing of usage of variable strategy is included in the benchmark for usage of each package's "own" ordering. Because for `RegularChains` and `ProjectionCAD`, strategy is provided by functions outside of the main CAD function, such calls providing strategy are unevaluated such that the timings for strategy are incorporated into the timings returned by CodeTools:-Usage to be recorded for the CAD benchmark. Some cyclic dependency occurs when using `ProjectionCAD`'s variable ordering to examine its own internal variable orderings. Designation of equational constraints in `ProjectionCAD` depends on overall variable ordering, which in turn depends on the designation of equational constraints (when using ECCAD or TTICAD, i.e. those functions that actually make any use of equational constraints). In this case the overall variable ordering chosen by strategy takes precedence, and equational constraints are chosen amongst clauses by use of a heuristic, but with the variable ordering passed to the heuristic for ECs passed as lexicographic. This is relevant as `cad_benchmark.mpl` attempts to let `ProjectionCAD` designate equational constraints by its own heuristic for such. In some sense, this parallels similar cyclic dependencies encountered in variable strategy for `QuantifierElimination` with respect to equational constraints, variable strategy, and Gröbner bases (Section 3.8), or the cyclic dependency of `IQER` selection strategy with variable strategy in VTS (Section 2.3.2).

Obviously there are a myriad of dimensions available to examine in terms of the data generated here, but the following figures examine various important ones among them. Each are generated from the generated `.csv` of benchmarking data for CAD associated with usage of the `bash` script `cad_testing.sh`, `CADBenchmark.csv`.

Figures 7-2 and 7-3 examine data about usage of each variable strategy offered by `QuantifierElimination` for CAD. The plots are expectedly similar, with the data being highly correlated. One again notes the variable ordering is generated separately before usage, so any one benchmark calculates exactly one set of projection bases of all orders. The calculation of any one benchmark is predicated on generation of the ordering, implying strategy must complete for that example. These figures tell an expectedly similar story considering the two statistics to examine are related. The classes of strategies identified by Figure 7-1 are somewhat reflected again in this data. Once again ECHeuristic is a small improvement, if any, over usage of the Brown heuristic, with the `RegularChains` heuristic falling shorter of this still. The same discussion from the projection benchmarking applies again to explain the potential small differences in the curves, considering `RegularChains`' heuristic has the same sentiment as Brown, with potential slight implementation differences. The performance of the "projection based" heuristics then falls into a better performance class. This differs from the case of performance in projection, where ndrr and sotd fell in the similarly worst performance class than the other strategies, although "greedy" allows for completion of the most examples, likely enabled by generation of the most orderings via Figure 7-1.

In light of Figure 7-1, we can deduce that the total case for CAD strategy implies a clear win for greedy. While ndrr and sotd similarly manage to make use of intricate low level data from projection to find similar performance to greedy out of the constructed CAD, their expense in terms of projection means that they cannot manage to be a win in total. While usage of the surface level heuristics line up in a fairly expected way, they

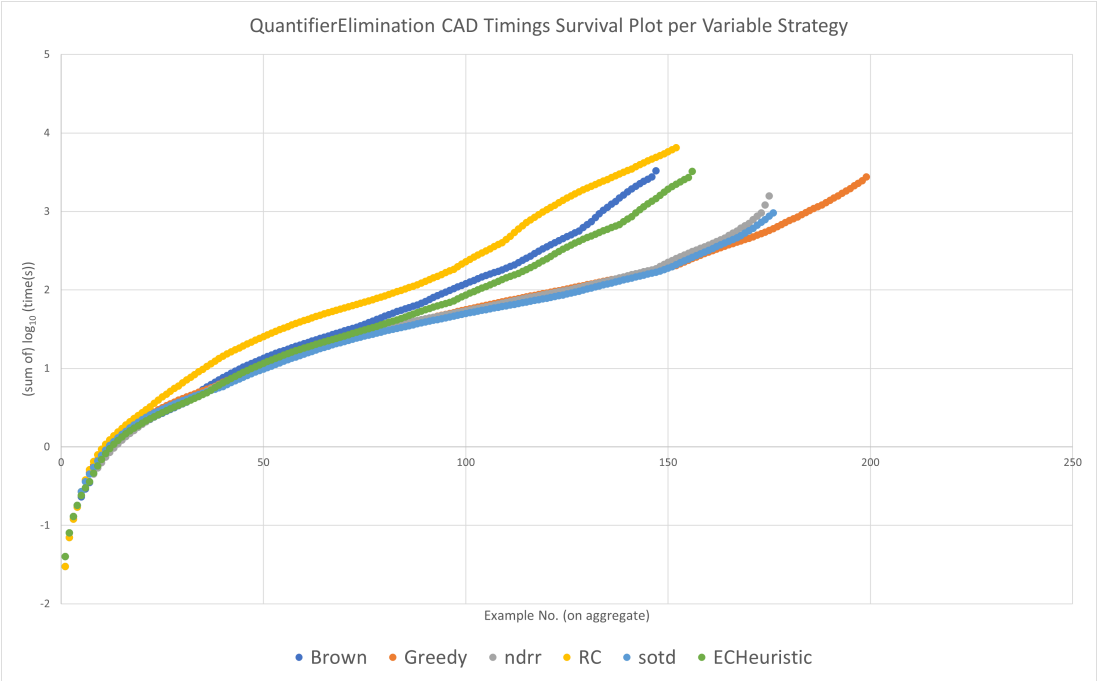


Figure 7-2: Survival plot for `QuantifierElimination` CAD per every strategy offered in terms of time for computation in seconds plotted logarithmically. The data here is on aggregate with respect to usage of equational constraints (`'UseEquations'`), and Gröbner bases are enabled where possible.

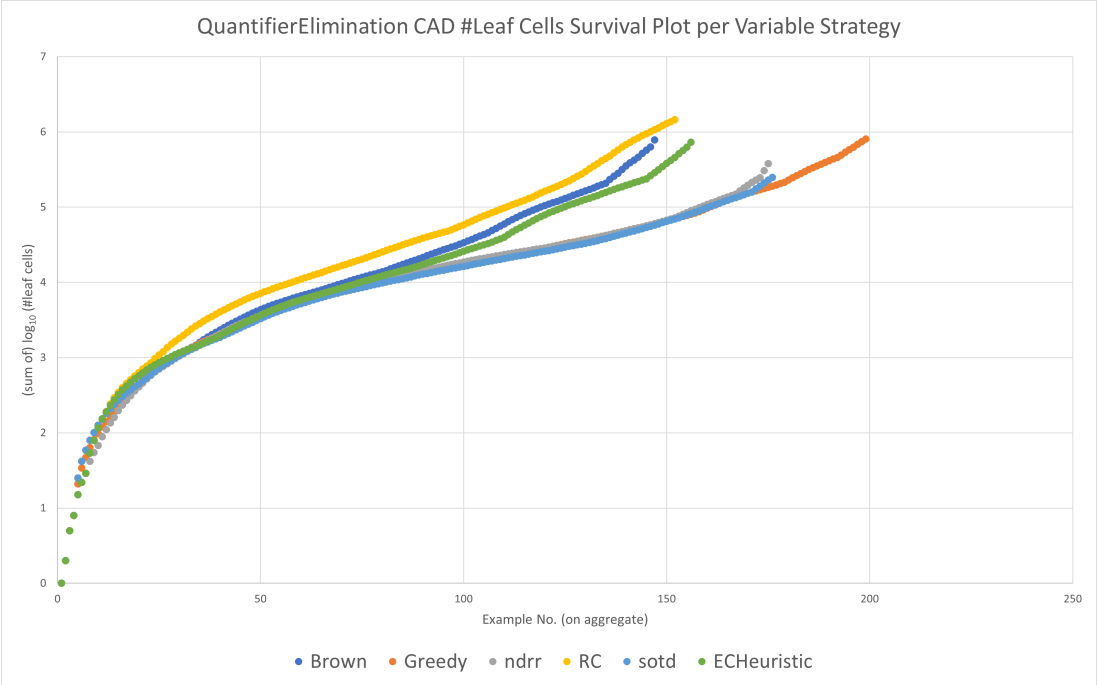


Figure 7-3: Survival plot for `QuantifierElimination` CAD per every strategy offered in terms of total number of leaf cells plotted logarithmically. The data here is on aggregate with respect to usage of equational constraints (`'UseEquations'`), and Gröbner bases are enabled where possible.

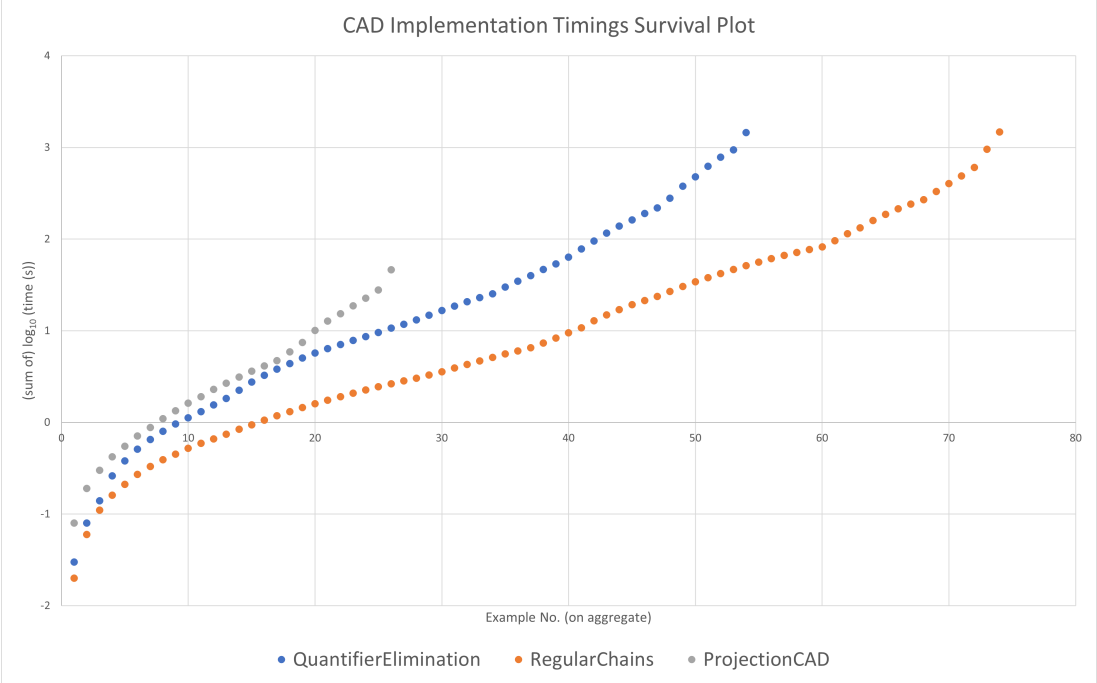


Figure 7-4: Survival plot per each benchmarked CAD implementation in Maple in terms of time for computation in seconds plotted logarithmically, where each implementation's "own" variable ordering was used. Only data from usage of single equational constraints is shown in terms of `QuantifierElimination` due to its completeness, and other options for `QuantifierElimination` are fixed as default (Gröbner bases are enabled).

end up being reasonably "middle of the road" in total, not managing to lift CADs that are as effective as greedy's, and the cost of projection being less than the fully verbose strategies, but again not beating greedy. A slight anomaly is the current incompatibility of the greedy strategy and Gröbner bases in terms of `QuantifierElimination`'s implementation. Gröbner is used for the CAD benchmarks to receive Figures 7-2 and 7-3, but is not used in the projection benchmarking of Figure 7-1. Again, this may flatter the projection benchmarks for greedy.

Figure 7-4 examines all three Maple packages together. `RegularChains` manages to be a more complete implementation, never failing in terms of something analogous to a "lifting failure". In total it is also most competitive. In particular there are no mathematical impediments in the methodology analogous to curtains or nullification. Additionally, it acts entirely incrementally, differing from the other packages in this sense. However, Figure 7-4 only examines the predicted best complete options for CAD offered by `QuantifierElimination`, i.e. usage of a single EC to exclude the possibility of low level curtains. ECHeuristic is used as the default variable strategy for `QuantifierElimination` here, although the previous analyses implies that the greedy variable strategy may be a better default strategy, which could improve the competi-

tiveness of `QuantifierElimination` here. Additionally this data only includes where Gröbner was used, hence the analysis of usage of GBs is a factor. In general, cell selection strategy is used to largely pointless effect below PartialCylindricalAlgebraicDecompose for `QuantifierElimination`, because of lack of removal of CAD subtrees when $m = 0$ as is the case here. This of course does not result in a large amount of overhead, and is far from a bottleneck. The curve of the survival plot for `ProjectionCAD` is likely heavily truncated due to nullification occurrences which force an error here, so there is not a wealth of data to examine for `ProjectionCAD` in terms of timings. The shapes of the curves imply that the Lazard projection is an improvement over usage of McCallum's, however one notes that `ProjectionCAD`'s default "own" strategy used here is a weighted usage of ndrr and sotd, and we know from Figure 7-1 that usage of these are essentially as costly as it appears. On paper, usage of these to calculate $\mathcal{O}(n!)$ full projection bases can be no less expensive as the same case for the Lazard projection with ECs. 43 instances of nullification implying lack of well orientedness occur out of 85 attempts at examples with 28 completions with respect to the relevant data. Of course, many of these instances of nullification impede `ProjectionCAD` from finishing examples such that the "survival rate" would otherwise be higher. For what a comparison is worth, low level curtains account for two failures to complete examples in usage of multiple ECs in usage of `QuantifierElimination`'s "own" variable strategy, but of course the contexts are entirely different, because curtains can only occur in the context of ECs. Further views to the frequency of curtains can be found in the QE benchmarking of Section 7.4.4.

Figure 7-5 examines the comparable metric in terms of the number of cells with a *true* truth value between `QuantifierElimination` and `RegularChains`. The number of cells with a particular truth value is assumed to essentially be commensurate with the number of cells in total, but we cannot generally examine the latter from `RegularChains`. In a sense this removes some element of quality of implementation and more precisely examines the output. Again, `RegularChains`' differing methodology results in fewer cells on aggregate. This analysis only examines usage of each implementation's default "own" variable strategy. In consideration of Figure 7-3, `QuantifierElimination` could become more competitive in such a comparison if "greedy" were used as the default ordering, where greedy appeared to be more competitive than ECHeuristic as a strategy in terms of the number of leaf cells yielded, mirroring the similar remark made with respect to timing performance.

Figure 7-6 examines usage of `QuantifierElimination`'s Gröbner bases in a boolean sense (as opposed to a comparison of differing monomial orderings used beneath). Minimization of the number of leaf cells produced (more broadly the number of cells in general) is a key goal of the particular usage of GBs in this package (Section 3.7.3), so this plot examines the total number of cells produced. This analysis shows that a few more examples complete owing to usage of Gröbner bases, with the curve showing at most a marginal improvement via the usage of GBs overall. While the curves are very close, individual cases from the data are more nuanced — Gröbner bases are occasionally helpful in terms of the number of leaf cells produced, and occasionally

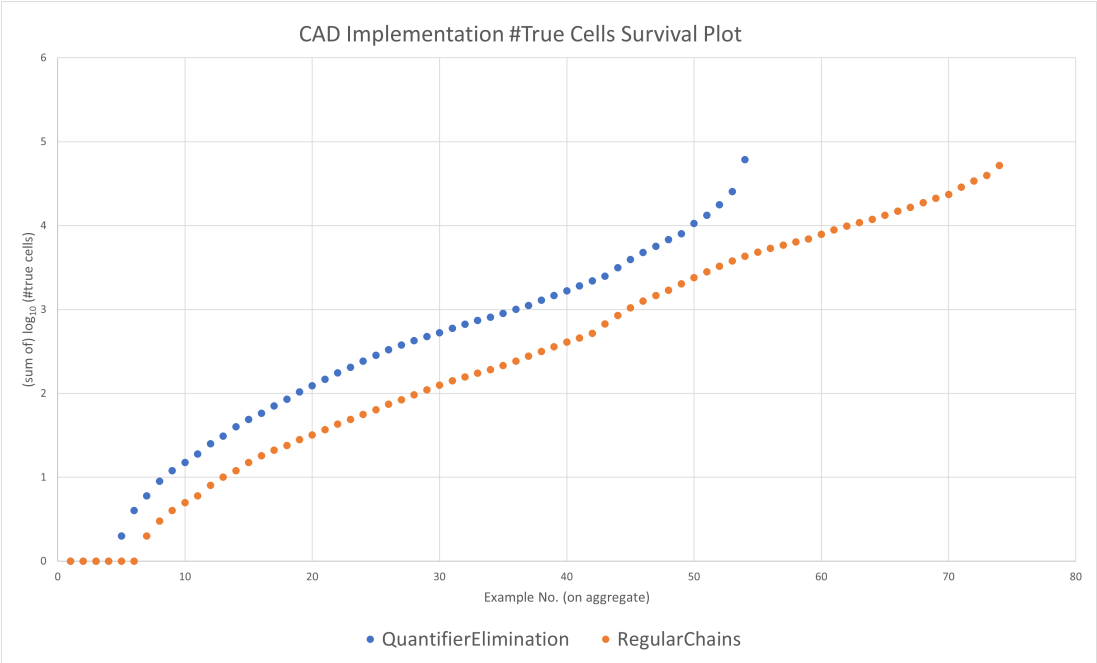


Figure 7-5: Survival plot per `RegularChains` and `QuantifierElimination` CADs in Maple in terms of number of leaf cells with a *true* truth value plotted logarithmically, where each implementation's "own" variable ordering was used. Only data from usage of single equational constraints is shown in terms of `QuantifierElimination` due to its completeness, and other options for `QuantifierElimination` are fixed as default (Gröbner bases are enabled).

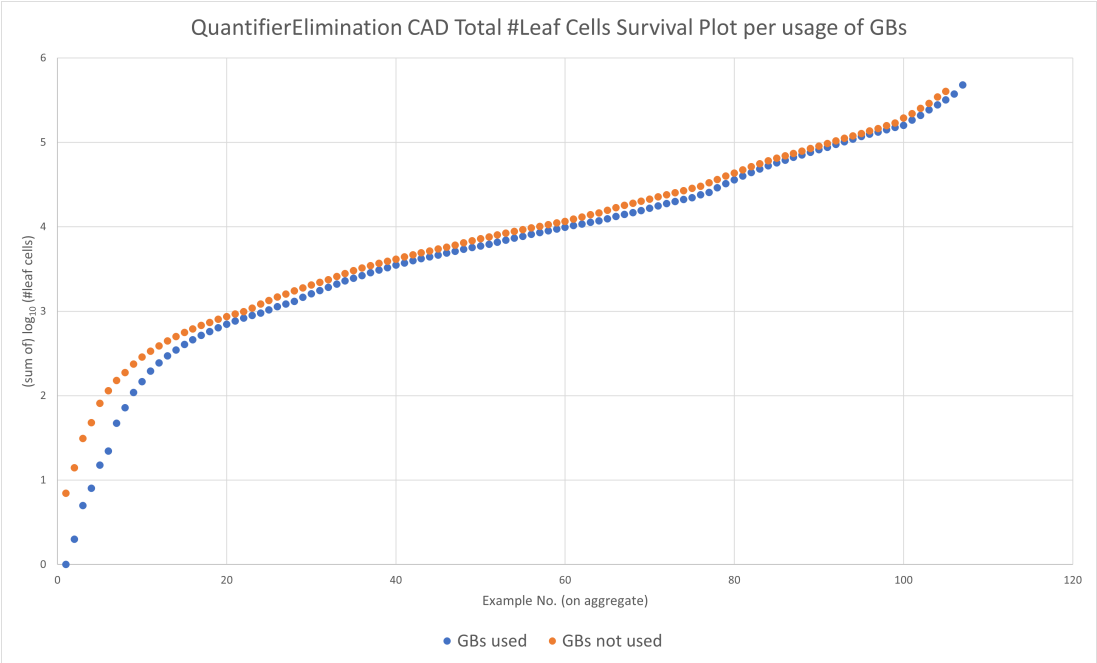


Figure 7-6: Survival plot for `QuantifierElimination` CAD per varying usage of Gröbner bases in terms of total number of leaf cells plotted logarithmically. The data here is on aggregate with respect to single and multiple equational constraints, where `QuantifierElimination`'s "own" variable strategy was used (i.e. ECHeuristic)

unhelpful. This is true for cases with usage of both a single EC and multiple ECs, and for cases outside of the "Cyclic" problems. The dataset is of course limited, and further the number of cases with applicable ECs even more limited. It is also possible that the Gröbner basis produced for a set of ECs could be equivalent to the original set of ECs amongst some cases. This analysis only incorporates data from usage of `QuantifierElimination`'s "own" variable strategy, and in reality it may be useful to examine data from every strategy, considering the number of leaf cells can be seen to vary significantly per strategy in Figure 7-3. Once again we note the "greedy" variable strategy is currently not applicable with Gröbner bases in this formulation. Ideally, a comparison against other monomial orderings could also be appropriate, including as part of a larger data set, perhaps those that always feature (identifiable) ECs. The case studies of Section 3.7.3 and 7.4.3 on individual examples still remain as claims to support that GBs in a Lazard projection CAD with ECs, especially with the bespoke monomial ordering presented here can be vastly helpful to a CAD using (multiple) ECs. Even outside of those cases which really are purely ECs, the data here supports that other more commonplace examples can benefit from these Gröbner bases. As for cases where the Gröbner basis was unhelpful, it could be that the resulting ECs at the lower polynomial levels of (3.5) are of a higher degree than they would be without this processing, hence more enumerable cells are produced at such levels, which hence has increasingly negative effects at higher levels yielding the leaf cells.

Lastly, Figure 7-7 examines the performance of varying usage of equational constraints via the option `'UseEquations'` beneath `QuantifierElimination`. The analysis is in terms of performance in terms of timings. The curves are surprisingly close, with `'none'` expectedly coming up short overall. More interestingly, the curves between `'single'` and `'multiple'` are close enough to be barely differentiable. Again, the data set is somewhat limited, and not every example will feature identifiable ECs, let alone multiple. Individual data points are again more nuanced, supporting that multiple ECs can of course be helpful. In fact, they can never be unhelpful in terms of the number of leaf cells to yield, unless they yield a low level curtain, which is always unrecoverable in this context of unquantified formulae. Immediately, one notes that usage of multiple ECs when multiple ECs are genuinely used in restricted projection implies more checking for (low level) curtains (Code Fragment 27, and in particular Algorithm 26) on the more enumerable pivot sets appearing in projection. In other words there can be pivots of more levels to check for a non zero Lazard valuation on, which may affect the timings to some extent. The data only includes that where Gröbner bases were enabled, which of course affects the usage of ECs as well, but there are various combinatorics in terms of dimensions to examine. The curve for multiple ECs is slightly truncated due to fewer instances of "survival" due to low level curtains. 14 examples failing due to low level curtains are identified amongst the examples that could otherwise appear in the aggregation of data here (there are 388 completions for `'multiple'`, and 398 for `'single'`, amongst 546 and 538 attempted examples respectively). Hence more examples process to completion under a single equational constraint, which is expected due to its completeness. This supports that usage of a single equational constraint is a sane default option for CAD. The curve for multiple ECs may support its use more

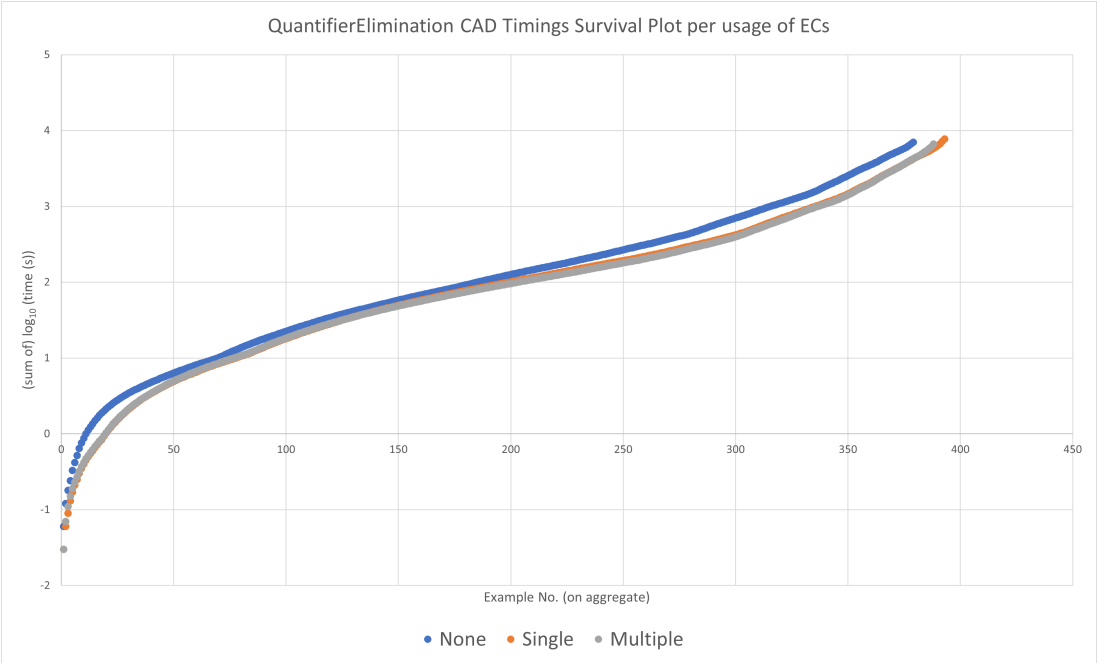


Figure 7-7: Survival plot for `QuantifierElimination` CAD per varying usage of equational constraints (`'UseEquations'` = `'none'`, `'single'`, or `'multiple'`) in terms of time for computation in seconds plotted logarithmically. Gröbner bases are enabled when relevant in every case, and the data is on aggregate with respect to variable strategies.

if those examples that fall to the wayside due to low level curtains were to otherwise complete, more quickly than the corresponding example under a single EC. In fact the existence of a low level curtain implies that multiple ECs were genuinely used in (semi-)restricted projection for such an example. The investigation as to efficacy of varying usage of equational constraints continues in QE benchmarking (Section 7.4.4), where usage of a single EC is compared against usage of multiple — this is also on a much larger dataset, albeit with a slightly differing context. The completeness of the approaches varies slightly there, due to the methodology of recovery from lifting failures.

There are some initially curious instances where usage of no equational constraints (`'UseEquations' = 'none'`) in usage of `QuantifierElimination`'s default "own" ordering can result in fewer cells. This is explained by `QuantifierElimination` discarding ECs ($A \leftarrow A \cup E$; $E \leftarrow \emptyset$) *before* usage of ECHeuristic to generate a variable ordering. In this way, usage of ECHeuristic essentially reduces to usage of the Brown heuristic on the set A, but this is not the same as usage of ECHeuristic when ECs would actually be present. Hence the ordering generated can actually differ, and in fact be more effective in the number of leaf cells produced compared to the case for usage of ECs with non trivial usage of ECHeuristic. Because ECs are not discarded when `'UseEquations' = 'single'` or `'multiple'`, the variable ordering generated with ECHeuristic is always the same in this case, and the number of cells produced in usage of a single EC is always at least the number when using multiple ECs.

In conclusion, the greedy strategy seems to be effective enough to warrant its promotion to the default variable strategy for CAD in `QuantifierElimination`, with some slight mitigations in terms of investigation of Gröbner bases in this context. `'UseEquations'` = single also seems to be a safe default option for usage of ECs in CAD for `QuantifierElimination`. The complications of low level curtains are expectable, and the opportunities to improve various aspects of methodology to deal with such are constantly highlighted in this work. The performance and rate of success of `QuantifierElimination`'s CAD is expected to be improved by development of various low level operations. The current projection & lifting methodology appears to fall short of `RegularChains`' differing methodology, but may be improved by variable strategy and further research on multiple ECs.

Usage of Gröbner Bases for Examples of Pure ECs

One briefly notes specific examples in this benchmarking exemplifying the extra performance from usage of Gröbner bases (Section 3.7.3). Many of the formulae are purely of equational constraints, i.e. a conjunction of purely equations. In particular this is true for the Cyclic-n problems, which are traditionally Gröbner basis benchmark problems in Computer Algebra. However, one can existentially quantify these formulae with all but one variable each to examine conditions on that variable such there exist real solutions to the equations (the examples are fully symmetric in terms of the variables). This is not the same problem as just generating all the solutions to the system of equations, where usage of CAD or even QE in general would be extremely verbose compared to a Gröbner basis approach not least due to the creation of open

geometry.

Regardless, we find that usage of Gröbner bases on these problems is clearly beneficial in terms of the number of leaf cells yielded and time taken to produce such. In particular these examples yield the case where $k = n$ in terms of Section 3.7.3, and with the solutions being zero dimensional the particular monomial ordering used coerces the desired triangular system (3.5), restricting propagation of equational constraints that produce spurious solutions hence yielding more numerous unnecessary cells. While Section 3.7.3 compares the case in terms of cells against usage of a typical monomial ordering from past literature, here we briefly examine `QuantifierElimination`'s usage of Gröbner bases against no usage of GBs. Figure 7-6 is the survival plot providing the full comparison in terms of cells from examples on aggregate just in terms of ECHeuristic as the variable strategy. In terms of the Cyclic-3 example from Section 3.7.3, we note the difference in terms of cells is 45 with GBs, and 181 without (`'UseGroebner'` = *true* and *false* respectively). This is via 6 and 11 projection polynomials respectively. If one is to use full CAD with the monomial ordering `plex`(c, b, a) to match past literature with identical other options, one also produces 119 leaf cells, identical to the example under QE from Section 3.7.3.

7.4.4 Quantifier Elimination

We benchmark various implementations of QE in Maple, and `QEPCAD B`. As usual the description of the background of the software is in Section 1.3. The list of implementations to benchmark, with particular options varying in terms of those below `QuantifierElimination` are the following:

- `QEPCAD B`, which requires a file to be redirected in defining an example in order to automate the process. It also has no intrinsic variable strategy, with the ordering having to be defined at input with the example. Hence we use the Brown heuristic in generation of the file to redirect in from Maple, `qe_to_qepcad.mpl` or `synrac_to_qepcad.mpl`.
- `SyNRAC`'s SyNRAC:-qe. The input semantics are similar to that of QuantifierEliminate in `QuantifierElimination`, albeit with the symbols `'All'` and `'Ex'` being used in place of $\forall$, $\exists$ as quantifiers, and `'Impl'` the symbol replacing `'Implies'`.
- `RegularChains`' SemiAlgebraicSetTools:-QuantifierElimination [47]. The input semantics are quite different, requiring a sequence of blocks of quantifiers using `All` and `Ex` followed by a boolean formula using `&and`, `&or`.
- `QuantifierElimination`'s QuantifierEliminate, QE by poly-algorithmic QE with
  - depth-wise traversal of the VTS tree for CAD,
  - breadth-wise traversal of the VTS tree for CAD, or
  - standard usage of VTS into CAD (`'HybridMode'` = `'whole'`).
- `QuantifierElimination`'s PartialCylindricalAlgebraicDecompose, QE by Partial CAD with the Lazard projection and equational constraints, where we vary usage of equational constraints in terms of

– single equational constraint (‘UseEquations’ = ‘single’),
– multiple equational constraints (‘UseEquations’ = ‘multiple’), which may lead to low level curtain errors.

An obvious competitor to QuantifierEliminate is SyNRAC, being similarly implemented in Maple, and implementing both VTS and CAD. When QE in SyNRAC by VTS yields a formula of entirely excessive degree, SyNRAC switches to CAD in a non poly-algorithmic sense. In terms of QuantifierElimination, we experiment with usage of QuantifierEliminate's ‘HybridMode’ keyword option to control usage of the poly-algorithm, either via ‘depth’ or ‘breadth’-wise traversal of the VTS tree, or by passing of the symbol ‘whole’ to disable usage of the poly-algorithm. When we receive the situation of all ineligible IQERs in an early block of quantifiers, ‘whole’ is implicitly always used as the poly-algorithm is inapplicable, so not all examples donating ineligible IQERs are amenable to the poly-algorithm. Additionally, one notes that formulae that are not Tarski formulae in the sense of Definition 3, i.e. those with irrational numbers always pass through directly to Partial CAD despite usage of QuantifierEliminate, as the methodology that can deal with Real Tarski formulae. When CAD is used in the poly-algorithm, multiple equational constraints are always used in projection, due to the poly-algorithm's attempt to globally avoid lifting failures from CAD. RegularChains provides QuantifierElimination below the sub-package SemiAlgebraicSetTools as of Maple 2020.1, using their methodology for CAD to achieve QE. RegularChains does not appear to accept Real Tarski formulae, being only amenable to quantified Tarski formulae. In other words, it will not accept radicals or RootOfs in quantified input. This only rules out one example from the QEExamples database, although it does not seem to reject the formula up front. In terms of PartialCylindricalAlgebraicDecompose, we experiment with usage of single vs. multiple equational constraints (‘UseEquations’ = ‘single’ or ‘multiple’), where only the latter will result in low level curtains, but perhaps introduce extra efficiency in examples via the lifting failure avoidance of Partial CAD.

Each function generates their own variable ordering for QE via whatever is default for that function. More generally, other than any varying options as are delineated above, every implementation always uses default options in terms of strategy etc. for every benchmark. Notably, ‘MaxVSDegree’ = 2 for QuantifierEliminate such that VTS can eliminate variables appearing up to quadratically. RegularChains' QE default options has the returned output formula for QE as a Tarski formula. There is an option to define the level of simplification for output [16], and it seems by default at least some level of simplification is deployed on the “cylindrical formula” gleaned from CAD. The underlying CAD methodology is always incremental, and by default “partial” with truth values to enable early termination for QE as is common. qe_benchmark.mpl is the Maple script to prepare the arguments to pass to the function to benchmark. The QE and economics databases are in format compatible with QuantifierElimination, but the SyNRAC database is in format compatible with SyNRAC, so in general we need to convert any one example to format amenable to the implementation to benchmark. Additionally examples may as well always be passed in prenex form to remove any instances of that conversion from timings. qe_testing.sh is the bash script iterating

over examples and packages as usual.

In order to benchmark `QEPCAD B`, it must be redirected a file defining the example. `qe_to_qepcad.mpl` is the Maple script to write the file for `QEPCAD B` to read for the QE or SyNRAC databases respectively. We use the Brown heuristic implemented from `QuantifierElimination` in order to provide `QEPCAD B` with a variable ordering for the example, which forms a part of this Maple script. The output from `QEPCAD B` can be redirected to a file such that we can deduce almost all the same data that we would be able to from the Maple implementations, sans the amount of memory used. `qepcad_test_qe.sh` is the `bash` script handling the `QEPCAD B` call with timeout and recording of relevant data. `QEPCAD B` is passed the option `+N8000000` to allow for more cells in the garbage collected space (4x that of default, so 8000000 garbage collected cells) as per basic documentation of `QEPCAD B`. `QEPCAD B` here produces output formulae as Tarski formulae rather than Extended Tarski formulae in the sense of Definition 33. This is the default for `QEPCAD` — the examples needing to be redirected in non interactively means that we have difficulty requesting an ETF at the relevant time in computation. If `QEPCAD B` is to fail to complete QE due to nullification (Section 3.7.2), then we do not restart using $PROJH$, such that we can compare against occurrences of curtains from `QuantifierElimination`. Examining a reason for failure for `QEPCAD B` to complete an example is difficult considering the generality of working with general text output, but one can always glean an exit code in these cases, recorded in the error column of the `.csv`. As always we must check for timeouts from Maple or `QEPCAD B`.

All that is left to do is to examine the survival plots produced from usage of every methodology above on the QE, economics, and SyNRAC example sets, which forms a total of 452 QE examples to benchmark. Figure 7-8 represents a survival plot per methodology in terms of time, created from the generated `.csv` file of raw data from the QE benchmarks, `QEBenchmark.csv`, which itself was generated using the `bash` script `qe_testing.sh`. Some calls for the Maple implementations are obstructed slightly by bugs in low level Maple routines, generally at the same rate. For `QuantifierElimination` these are usually failures in floating point type operations within CAD, that are in some sense lifting failures, although not categorised as such due to their impending resolution in future releases of the software.

Figure 7-8 evidently identifies `SyNRAC` as a very competitive implementation. The Maple native `RegularChains` falls not too far short, with QE being a relatively recent addition to the package, based on their CAD methodology. `QEPCAD B` is the only software tested outside of Maple, and performs admirably. One notes timings may not be directly comparable with those generated specifically by Maple, but the usage of timeouts are always equal, and so `QEPCAD B`'s "survival rate" is directly comparable. Of course the low level operations used by `QEPCAD B` are completely disjoint from any of those used by the Maple packages. The shape of the curve for `QEPCAD B` largely keeps pace with that of `SyNRAC`, with a sudden sharp rise toward the later end of the curve, implying impressive performance on smaller examples, but some difficulty on the much larger ones. The considerations of `QEPCAD B`'s interface is otherwise well discussed elsewhere in this section.

The `QuantifierElimination` methodologies all implemented for this project ev-

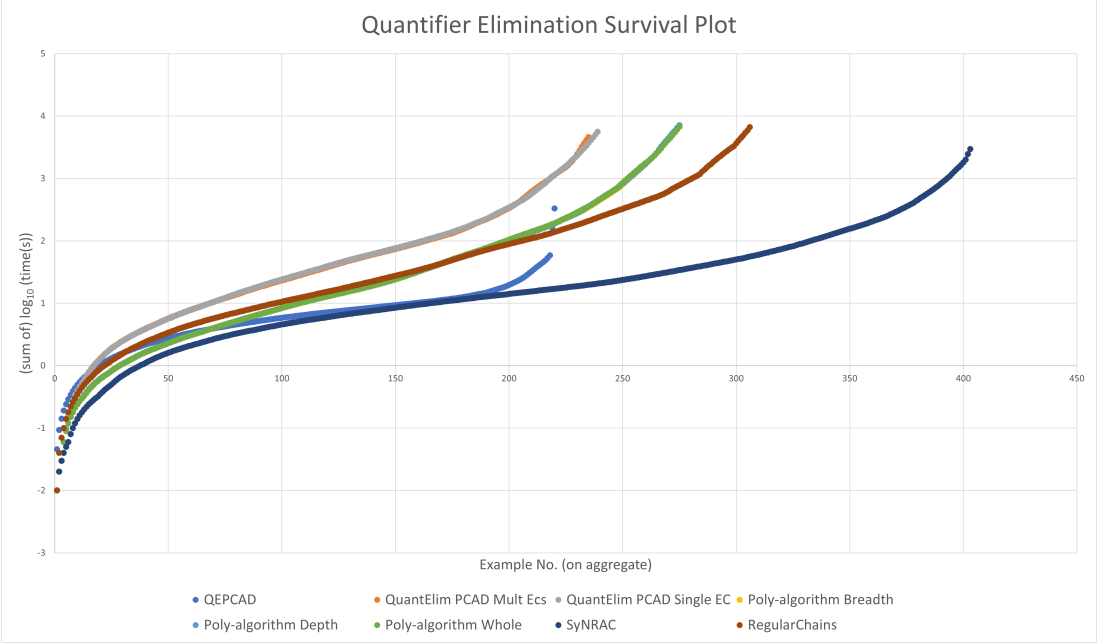


Figure 7-8: Survival plot for various Quantifier Elimination implementations with variation of certain delineated options, against time (s) logarithmically.

idently provide key comparison, being from the same author. Firstly, PartialCylindricalAlgebraicDecompose's performance is close under differing usage of equational constraints (`'single'` vs. `'multiple'`). Their usage can clearly be identical for examples that do not feature ECs, so some broad similarity of the curves is to be expected, but in fact exactly the same number of examples are completed between these two approaches. There are a number of examples that fail to complete for multiple ECs due to unrecoverable low level curtains, but on the other hand multiple ECs manages to complete roughly the same number more over usage of a single EC due to the extra performance introduced by multiple ECs when unimpeded by curtains. In total the two approaches manage to be surprisingly close. One consideration is that in the case for `'UseEquations'` = `'multiple'`, when multiple ECs are genuinely identified, CAD lifting must check for curtains more frequently, and in particular on larger (lower level) projection polynomials, which attributes cost. In addition Partial CAD must generally attempt avoidance of more low level curtains as a result. The data suggests `'single'` should become the default option for usage of ECs beneath PartialCylindricalAlgebraicDecompose due to its mathematical completeness. Usage of multiple ECs can remain a viable "Monte-Carlo" option for examples with highly numerous ECs where the user finds an attempt at QE with usage of a single EC too slow, due to the existence of examples where multiple ECs beats its counterpart. Gröbner bases are always used to preprocess ECs here, but the previous analysis suggested this is sometimes a detriment, which could complicate this analysis slightly. Meanwhile, the poly-algorithm via QuantifierEliminate outperforms the pure Partial CAD approaches. A sizeable portion of the example set tested on are the economics QE problems via

the economics database. These examples are largely all linear per variable, and usually highly enumerable in the number of variables. These are evidently cases where VTS is understood to outperform CAD, with CAD suffering from the number of variables especially. That being said, PartialCylindricalAlgebraicDecompose can sometimes make good use of single or multiple ECs to solve some of these examples, which quite often feature many ECs. Curiously, this includes cases where it can solve examples where VTS fails to. Use of PartialCylindricalAlgebraicDecompose is surprisingly competitive on linear examples considering a pure CAD approach compared to `RegularChains`, and more broadly the `QuantifierElimination` functions seem reasonably competitive on examples that are largely linear or quadratic, whether it be due to usage of VTS or surprising cases where CAD solves such examples despite enumerable variables (but potentially many ECs). Use of the poly-algorithm on the economics examples always falls to usage of pure VTS without CAD, with VTS being able to traverse problems that are purely linear without assistance. In terms of more standard examples, comparison of the poly-algorithm and Partial CAD varies. Infrequently, usage of VTS appears to be vastly unhelpful before CAD in any context. Lack of strong simplification for VTS is a constant lamentation always highly identified as to when VTS introduces poor performance. Other cases are more nuanced, such as examples which feature a high number of ECs. Most particularly, one notes that the poly-algorithm performs worse than Partial CAD approaches on the Cyclic-n problems, which are entirely of ECs. Usage of VTS for QE never uses Gröbner bases for preprocessing, but the Partial CAD approaches are using the bespoke Gröbner basis preprocessing (Section 3.7.3) by default, because there is a clear view to optimising the projection process. VTS' poor performance here is more interesting in terms of the fact the Cyclic-n problems are linear in terms of any one variable, albeit not in terms of total degree. In other cases where ECs do not appear as prominently, but VTS still appears as unhelpful, one notes quadratic elimination via VTS can induce degree bloat. This is where investigation of variance of the keyword option `'MaxVSDegree'` to restrict VTS to a maximum of linear elimination could inform this discussion better. One notes that usage of PartialCylindricalAlgebraicDecompose is functionally equivalent to usage of QuantifierEliminate with usage of the option `'MaxVSDegree'` $= 0$. The previous sections identified that "greedy" may be a better default variable strategy for CAD than ECHeuristic as was used here, suggesting potential performance increases for `QuantifierElimination` would be possible under this strategy.

In examination of the poly-algorithmic methodologies, we note that all the methodologies default to usage of multiple ECs under the hood whenever CAD is concerned, but this has never resulted in low level curtain errors when the poly-algorithm is used in a non trivial sense ("Solotareff-3" from the QEExamples database immediately resorts to full Partial CAD immediately due to excessive degree). Usage of the poly-algorithm in a non trivial sense implies that CAD acts upon real space in fewer variables, perhaps restricting the opportunity for low level curtains to arise. The case studies of Section 7.3 are usually in few variables after action of VTS. One notes that the incremental methodologies for CAD implemented in `QuantifierElimination` pay careful attention to identify curtains in the context of incrementality, such as Algorithm 52. Because the poly-algorithm only acts in the context of a last homogeneous block of quantifiers,

it may be the case that this maximises opportunity for "local" avoidance of curtains within CAD calls, although such examples usually feature free variables. Additionally, one notes that the poly-algorithm ignores lifting failures beneath `IQER`s until they are deduced to be entirely blocking to deduction of QE due to the search for meaningful truth values, so it is more likely QuantifierEliminate times out in searching for meaningful truth values than survives to reraise such an exception. While curtains never cause the poly-algorithm to error out, there are instances where CAD below the poly-algorithm successfully recovers from curtains, case studies for which are delineated in Section 7.4.4. These case studies support the methodology used to avoid and recover from curtains, while also furthering the case for improvement of such.

There is currently no way to disable the avoidance and recovery of lifting failures within CAD in `QuantifierElimination` without loosening early termination criteria to the extent that performance would be significantly impeded, so it is difficult to examine to what extent such failures are dealt with in the context of QE, especially with respect to curtains. We only know when we certainly had to error out of an example, because `QuantifierElimination` is true to Code Fragment 35 in the sense a mathematical error about a Lazard curtain is always reraised in preference of any other lifting failure if any others exist, when QE could not be reliably deduced. Hence the benchmarking data always reflects whenever an unrecoverable curtain is found. Hence we are sure to have identified 18 examples attributing low level curtains amongst 452 total attempts at examples in usage of multiple equational constraints for PartialCylindricalAlgebraicDecompose. This is of course not a maximum of 18 low level curtains, due to the presence of timeouts, or other lifting errors impeding computation on examples that could otherwise yield such curtains. Multiple ECs attributes 4 fewer completions of examples than usage of a single EC under PartialCylindricalAlgebraicDecompose. The 18 manifestations of low level curtain errors imply that multiple ECs could easily attribute more completions than usage of a single equational constraint if the low level curtains could be overcome — both projection and regulation lifting must complete for the CAD to raise such an exception, due to the methodology that attempts recovery as far as possible.

The options for `'HybridMode'` beneath QuantifierEliminate to vary usage of the poly-algorithm are also similarly close, to the extent they are barely differentiable in the survival plot. Again, the curves are expected to be similar in shape, even more so than the differing cases for PartialCylindricalAlgebraicDecompose. This is because the poly-algorithm is often inapplicable in a non trivial sense, either because CAD is not required due to no ineligible `IQER`s, production of an obstructive ineligible `IQER` in an early block of quantifiers, or even possibly the rare case of completion of the VTS tree for a last block of quantifiers with just one ineligible `IQER`. Hence in the majority of cases QuantifierEliminate acts in exactly the same way, which can be seen by examination of the benchmarks. The same number of examples are completed between the three available options for QuantifierEliminate, even while individual cases can differ — the same number are completed, but this is not owing to the same set of examples. The aspects of incrementality of the genuine poly-algorithmic approach of course incur some overhead in comparison to the "whole" methodology. Variable strategy & the

resulting orderings can vary significantly between usage of the incremental depth or breadth-wise methodologies and the "whole" methodology, and we note usage of the former can fix a poor ordering for later incremental CAD operations, which is not accounted for by the existing "poly-share criteria". In fact, strategy in general could pay more attention to what is the "poly-share criteria", given traversal in terms of height is prioritised before evaluation of the criterion. Variable strategy in the "whole" methodology is more unconfined, being purely a means to an end (Section 4.1). The intrinsic simplification provided by the "whole" methodology often results in building much smaller QE output formulae. The usage of ECs under incremental projection may be imperfect when we cannot be sure of reusing the same EC in restricted projection (line 38 of Algorithm 50). To observe differences in variable ordering, one can see that often the quantifier free output from usage of the "whole" methodology often owes to a different ordering — the conjunctions in CAD QE output owing to the cell descriptions follow the same ordering as that used for the unquantified variables. Further, via the case studies of Section 7.3 we note that usage of depth and breadth-wise traversal in the poly-algorithm can be identical. Where depth or breadth-wise traversal performs well, we note that reuse of boolean structure including ECs seems to be a particular boon. The benefits of usage of poly-algorithmic QE remain with the canonicality of the approach allowing for evolutionary methods and production of witnesses in the fully homogeneously quantified cases, paired with the opportunity for further research to improve the methodology itself and to cater the package further for the case of `QF_NRA`. Once again, there are non trivial considerations in terms of simplification that may support the case for the poly-algorithm further as well, due to simplification on the formulae for individual `IQER`s being more canonical (Section 4.3). The poly-algorithmic approach is essentially never a detriment, hence being the virtually best approach for QE offered by `QuantifierElimination`, especially in light of the features offered.

It is not easy to tell from this investigation exactly which examples can yield a situation where the poly-algorithmic methodology has scope to differ from the other approaches in a non trivial sense — in particular from the `'whole'` methodology. Quantifier free output for a benchmark being an extended Tarski formula (in particular featuring real algebraic functions) is not a sufficient identification — there must be at least two leaf `IQER`s, including at least one ineligible `IQER` occurring in elimination of a last block of quantifiers such that the genuine poly-algorithm (with depth or breadth-wise traversal) has scope to differ from the `'whole'` methodology. A function that would directly allow for examination of the frequency of this situation is noted as important canonical further work (Section 8.3) — a function that performs pure VTS with no CAD below `QuantifierElimination`, which would be incomplete in terms of QE due to degree limitations. Various examples of non trivial usage of the poly-algorithm have been identified in individual case studies, Of course instances where benchmark timings for the methodologies differ to a large extent exemplify examples where the methodology differs. Beyond this, the need for time outs of course limit the extent to which differences in methodology can potentially be identified — at least by failure to complete propagation of VTS. Furthermore, there is a distinction between identification of an ineligible `IQER`, and a case where this impedes QE, due to meaningful truth values of `IQER`s.

There are at least three instances of usage of Algorithms 32 and 34 to attempt to recover from level $n-1$ curtains in the context of PartialCylindricalAlgebraicDecompose. The first arises via the example "Simplified Putnum" from the QEExamples database, viz:

$$\exists a \exists d \; a^2 + d^2 - 4\,dy + 4\,y^2 - 1 = 0 \wedge a^2 - 4\,xa + d^2 + 4\,x^2 + 20\,a - 40\,x + 91 = 0.$$

We briefly explore this example as a minor case study of an instance of curtain decomposition in QE. Both usage of single and multiple ECs in PartialCylindricalAlgebraicDecompose fail on this example due to failure to evaluate a deduced lifting constraint (which has irrational coefficients) during root isolation within Algorithm 34. The obstructive lifting constraint differs per the value of `'UseEquations'`. Failure to evaluate a lifting failure with irrational coefficients is a precedented type of lifting failure (item 4, Section 3.7.2). Because this is the only situation where lifting constraints are used as standard within QE in any context, this notifies that curtain decomposition was attempted. Further, the curtain decomposition here does not reduce to anything trivial in the same way as the majority of the case studies of curtain decomposition for full CAD (Section 7.2). More specifically the generated set of univariate polynomials after set difference with projection is not empty. It is also obvious that the single round of regulation lifting to attempt to recover from any point curtains that could be deduced via confidence was also insufficient for recovery. There are initially 18 cells with lifting failures, all of which are due to curtains. Amongst the 18 curtains, all are of level $n - 1 = 3$. The confidence criteria from Lemma 59 allows us to classify 6 of them as point curtains. This is the case for both single and multiple ECs. Regulation lifting on those 6 point curtains does not allow us to achieve QE by propagation of truth values to ignore the other 12. Hence the other 12 enter recursive curtain decomposition, and it is unclear if they cannot be identified as point curtains due to lack of required neighbour cells, or the latter clause in Lemma 59. As a result of lack of full confidence to deduce point curtains, it is conceivable that we enter curtain decomposition on more than just non point curtains here, which anecdotally results in a large number of new cells during this decomposition. One notes these level 3 cells are within existentially quantified space immediately below unquantified space, i.e. their neighbours should always exist because every level 3 cell should exist, being the child of a cell in unquantified space. Considering the lack of other lifting failures in this example, it is likely the example would finish if CAD was able to overcome being unable to evaluate the deduced lifting constraints.

A second known example of curtain decomposition in the context of PartialCylindricalAlgebraicDecompose appears via "kanazawa2014-Ri-1-m-2" (i.e. the second example under such a file name) from the SyNRAC QE database under PartialCylindricalAlgebraicDecompose with single or multiple ECs. In the case for a single EC, there are 8 curtains amongst 3 other stored lifting failures. In the case for multiple ECs there are just 4 curtains with no other lifting failures. In any case no curtains can be identified as point curtains via confidence criteria. Again, the recursive process attributes a significant time investment to the extent that the example fails to complete within the time limit with usage of single or multiple ECs. This is of course a corollary of the set

difference at level 1 not yielding the empty set, in contrast to the majority of the case studies in the case for full CAD.

In both cases for these examples, deduced lifting constraints often feature real algebraic numbers making root isolation more difficult, in the first case to the point of failure. While it is not benchmarked here, usage of no equational constraints (`'UseEquations' = 'none'`) actually allows these examples to finish under PartialCylindricalAlgebraicDecompose within the allotted time out, with less time being expended than in the case with even multiple ECs. The examples also finish successfully if usage of GBs is disabled with any usage of ECs, implying any curtains can be avoided or not produced in this case. Lastly, usage of VTS to "preprocess" the examples allows the poly-algorithm with any options to complete these examples successfully. Further inspection of the details of the curtain decomposition in the context of QE for these examples can be found via Maple userinfo by specifying `infolevel[ PartialCylindricalAlgebraicDecompose ] := 10`. These cases for curtain decomposition in QE by Partial CAD are non trivial, in contrast to the analogous case studies of Section 7.2 investigating usage of Algorithm 30. In other words, the level 1 projection basis created in recursion is not a subset of the original level 1 projection basis in these cases. The cases for recursive curtain decomposition in PartialCylindricalAlgebraicDecompose for QE are relatively rare, with there being two identified amongst a total of 452 examples. Lastly, there is an additional example where CAD recovers by lifting point curtains. The third and final known example of curtain decomposition beneath PartialCylindricalAlgebraicDecompose appears via "kyoto1999-Ri-6-m-1" from the SyNRAC QE database. Poly-algorithmic QE via QuantifierEliminate is equivalent to pure Partial CAD here, due to the root `IQER` being ineligible in this case. Here, curtains being identified as point curtains via confidence criteria allows us to lift to success without entering recursive curtain decomposition. The examples specific to the poly-algorithm discussed below also work similarly, where the discussion about lifting of point curtains to success continues.

There are two instances of usage of Algorithm 32 that can be observed via non trivial usage of breadth-wise traversal in poly-algorithmic QE — "yozemit2016-1-Ri-3-s-1", and "kyushu1999-Ri-5e-m-2"from the SyNRAC QE database. As usual these cases can be investigated via `userinfo` with high values of `infolevel`. In these cases, we enter Algorithm 32 for at least the first instance of Partial CAD (via QEPCADL) on an ineligible `IQER` of non trivial level. For both examples, we immediately recover from all lifting failures that are present by identifying curtains as point curtains via confidence criteria, and then subsequently lifting them to complete QE without entering Algorithm 34. In other words the first instantiation of regulation lifting at line 4 of Algorithm 32 is sufficient to deduce QE on the first `IQER` in these cases, despite exceeding the time limit in attempting to solve subsequent `IQER`s in the "global" QE call. In the first instance listed here, Algorithm 32 is called to success in terms of lifting point curtains to solve several further `IQER`s (via QEPCADL, i.e. without repurposing of a CAD) as well. In total, this is good news, as we obtain examples where identification of point curtains via confidence criteria can allow us to avoid a significant amount of work. We once again note the discussion which identifies that once a point curtain has been identified and lifted, or else a curtain cell has undergone further decomposition, it will no longer

be identified as a curtain by the incremental processes of `QuantifierElimination`, which is helpful in the highly incremental contexts of CAD such as the poly-algorithm (Section 4.1). This is likely the case as to why Algorithm 32 need only be called once beneath poly-algorithmic QE for the second example referenced here. Again, the cases for level $n-1$ curtain recovery below the poly-algorithm in a non trivial way come at a rate of 2 cases out of the total 452 benchmarked. A reason for Partial CAD in any context more rarely requiring recovery from level $n-1$ curtains than the case for full CAD could be that usage of Partial CAD restricts the scope for curtains to occur, because we do not brazenly construct every level n leaf cell. More generally the more non uniform construction of the CAD reconciling with exactly the same frustrations that impede identifying point curtains (e.g. discussion preceding Lemma 59) may mean that level $n-1$ curtains less frequently offer the opportunity to impede QE over more obstructive lifting failures that can occur earlier in construction of the CAD.

In conclusion, `SyNRAC`'s performance, and the poly-algorithmic approaches performance over PartialCylindricalAlgebraicDecompose imply that usage of VTS is still at least of interest, if not usually a win especially whenever particular linear examples are concerned, such as those from economics. With respect to `QuantifierElimination`, there are several identified factors to improve in terms of the implementation, most particularly with respect to VTS, such as simplification and the handling of universal quantifiers. The refactoring of these elements is unlikely to make the poly-algorithm's performance worse relatively to pure Partial CAD, but may change the balance of performance between the offered poly-algorithmic methodologies, and improve performance of these as a whole. The occasional increases in performance offered by usage of multiple ECs for pure Partial CAD and the confirmation that there exist examples where low level curtains impede completion of QE validates "completing" multiple ECs in the same manner as the completion of single ECs by [56] as useful further research. Additionally, and not orthogonally, methodology arising from solution of Open Problem 62 can mitigate low level curtains further, where multiple ECs clearly has some benefit over a single EC while still being impeded by these mathematical obstacles. Incorporation of propagation of truth values to remove subtrees of QE in such a more complex methodology could certainly assist Partial CAD to ignore low level curtains further than the current implementation, but would never "complete" multiple ECs with the Lazard projection. Further development of the low level elements of the CAD implementation in `QuantifierElimination` will mitigate the non mathematical types of lifting failure delineated in Section 3.7.2, and similar developments will also improve the performance of `QuantifierElimination`'s CAD when used in any capacity (Section 3.4.1). Investigation of `QuantifierElimination` in more "SMT-like" contexts, including actual SMT over the theory of real linear or non linear arithmetic will inform better discussion and evaluation of the greedy strategies implemented with a view to searching for meaningful truth values.

7.5 Comparisons of Input & Output

CAD in `QuantifierElimination` makes use of real algebraic numbers and real algebraic functions in the context of definitions 30 and 31 respectively, both internally and for output. Cell descriptions in `QuantifierElimination` intend to use real algebraic numbers where possible, else real algebraic functions (Section 3.4). The presence of RootOfs means there is some similarity between cell descriptions yielded by `QuantifierElimination` and those gleaned from usage of full CAD in `RegularChains` or `ProjectionCAD` using the keyword option `'output'` = rootof, but neither use interval indexed RootOfs in terms of algebraic numbers, instead using "complex" indexing via `'index'` = `'i'` for $i \in \mathbb{N}$ whenever RootOfs are requested for output. Complex indexing orders roots of polynomials in terms of "principal branching", which in particular may not correspond with the ordering of real roots with respect to the real line. Meanwhile, for `RegularChains` and `ProjectionCAD` the analogy of a real algebraic number is a regular chain, as is the namesake of the former package and underlying technology of both. The usage of real algebraic numbers as RootOfs indexed by intervals for output is intended to provide the user with as much information as possible, including the sign and approximate rational values of the root, although the rational numbers are usually of many digits because they are inherited from use of real root isolation under the hood, where the isolating intervals produced by RootFinding:-Isolate are of dyadic rationals. Reducing the Maple global value `Digits` feeds through to the precision used in real root isolation under `QuantifierElimination`'s CAD. isolateRootsOf uses a starting precision of `Digits`, and in general anything involving floating point operations such as root refinement starts with a precision as a function of `Digits`. An unnecessarily low value of `Digits` makes much of the floating point oriented operations very slow, such as root refinement and isolation (one may require much more root refinement in this case due to the coarseness of the isolating intervals produced). Some operations on real algebraic numbers in Maple are not highly well supported, and in particular may fail on highly nested real algebraic numbers, which is indeed one of the lamentations of Section 3.4.1. Additionally, RootOfs in terms of real algebraic functions are indexed with "real indices", i.e. `'index'` = `'real[i]'` (Definition 31). These have even more rudimentary support, but the intention of their usage in `QuantifierElimination` is entirely cosmetic to describe a `CADCell`, and as a consequence quantifier free output of QE.

`QuantifierElimination`'s variable strategy is always handled via keyword option, whereas `RegularChains` may require such to be outsourced by another function in the same package where CylindricalAlgebraicDecompose is concerned, but QuantifierElimination beneath `RegularChains` automatically calls its own variable strategy under the hood. While `QuantifierElimination`'s variable strategy is definable by providing the name of a strategy to use where CAD is concerned (Section 3.8), however an ordering can be forced by providing a list of variables as override, which is checked for validity. `SyNRAC` offers no obvious options for variable strategy, or indeed any options at all. `QuantifierElimination` offers a wealth of options related to QE or CAD where either or both are relevant. `QuantifierElimination` also offers the `QuantifierTools` package to assist with understanding Tarski formulae (Chapter 6).

`RegularChains` input for QE requires a sequence of blocks of quantifiers (one can group variables quantified in the same block by a list) followed by a formula. Usage of expression sequences in this way is often confusing for users, not being one object, but able to be assigned to variables, but not amenable to usage of the "op" command to iterate across it as a DAG (Directed Acyclic Graph). Input semantics for `SyNRAC` for QE are `QuantifierElimination` are similar, except for the fact that usage of the symbols `exists` and `forall` for `QuantifierElimination` typeset as $\exists$ and $\forall$, while `Ex` and `All` in `SyNRAC` do not. `SyNRAC` and `QuantifierElimination` share usage of the inert And and Or operators in Maple, which also typeset nicely, as opposed to `RegularChains`' operators. Curiously, `SyNRAC` uses the symbol `Impl` instead of `Implies`, the latter of which typesets well. `QEPCAD B` being interactive command line software is accessible, but lacks much of the quality of life features offered by use of any of the Maple packages. Obviously one can attribute much of this to its age and independence from integration within a Computer algebra package. Being an implementation of QE by CAD, its QE output with its version of extended Tarski formulae [12] is comparable to that of usage of PartialCylindricalAlgebraicDecompose, with real indexing of roots of polynomials making an appearance.

CylindricalAlgebraicDecompose's non optional input is highly coercible, being able to take sets, sets of relations, sets of equations, or a formula, always attempting to deduce ECs in an intelligible way in every case. PartialCylindricalAlgebraicDecompose performs QE on a quantified formula, but also produces a `CADData` object when an unquantified formula is passed, such that one can examine all leaf cells produced in the context of Partial CAD, including their truth values. One can deduce all leaf cells holding various truth values, or all leaf cells that may not be of level n due to evaluation of truth values on lower level cells making stack construction on such inapplicable. `RegularChains` and `ProjectionCAD` both take input that can be a list of polynomials, or something equivalent to a semi algebraic system in disjunctive normal form, via lists of relations and not via boolean operators.

`QuantifierElimination` is able to produce witnesses to prove truth of a fully existentially quantified formula or falsity of a fully universally quantified formula. This is not the case for the other Maple packages, or `QEPCAD B`. `SMTLIB` technically provides such functionality for unquantified formulae passed in conjunctive normal form as formulae for SMT (in this context over the theory of real arithmetic). Hence stripping a fully existentially quantified Tarski formula of all quantifiers and passing to `SMTLIB` provides similar functionality to passing the existentially quantified formula to a QE function from `QuantifierElimination` and requesting witnesses. SMTLIB:-Satisfy currently uses `Z3` [75] as its SMT solver under the hood, and returns a set of witnesses (as a Maple set) on satisfiability, or NULL if the formula is unsatisfiable. `QuantifierElimination` is able to produce as many witnesses as possible by loosening early termination criteria in usage of QE in any context via the keyword option `'eagerness'`. This option is by default set at 3, but reducing it will enable more traversal of the VTS or CAD trees alike (including in a poly-algorithmic sense), and will therefore attempt to yield more meaningful leaf `IQER`s or `CADCell`s for witness production.

Quantifier free output of QE by QuantifierEliminate where VTS can yield the an-

swer alone is a Tarski formula, albeit sometimes an egregiously unsimplified one, that without strong simplification may even be equivalent to *true* or *false* in a candid sense. SyNRAC:-qe's output is far more simplified, with simplification being a focus of the developers in [40], but in some cases still cannot (or does not attempt to) deduce the candid equivalent of an output expression technically equivalent to *true* or *false* (such as $c^2+c+1=0 \equiv \mathit{false}$). Quantifier free output of QuantifierEliminate where CAD is used to define the quantifier free output is always candid in terms of individual atoms, due to CADs with truth values being "their own simplifier" in this sense. This is also true of other implementations using CAD for output, and so `SyNRAC`'s failure to deduce $c^2+c+1=0 \equiv \mathit{false}$ is likely due to pure use of VTS and not CAD. While individual atoms in output from CAD are always candid, the overall formula may not be, and `QuantifierElimination` could perhaps benefit from simplification in terms of cylindrical formulae — some relevant work in this area is [16]. As a result of simplification of cylindrical formulae in this way, `RegularChains`' QE by CAD manages to have output which is closer to candid. Where the poly-algorithm is concerned, non candid atoms can be displayed in output owing to `IQER`s that did not undergo processing by CAD in any way (Section 4.3). The implementation of QE in `QuantifierElimination` in any context never factors out atoms completely of free variables. For VTS, this is no issue, as VTS intrinsically ignores such atoms. However, for CAD, this is an issue — as can be seen with usage of PartialCylindricalAlgebraicDecompose on the QE example "Hong-90" from the QEExamples database, where CAD produces an unsimplified "cylindrical" formula equivalent to $r+s+t=0$, which is the unquantified atom from the original formula, and also the quantifier free equivalent of such. VTS from the poly-algorithm, or the other packages correctly identify $r+s+t=0$ as the simplest answer. However ideally `QuantifierElimination` would just factor such unquantified atoms out of the input formula Φ to process in all instances, especially before QE by Partial CAD.

Whenever CAD is used to achieve QE in `QuantifierElimination`, the formula is certain to be an Extended Tarski formula, which is a supertype of a Tarski formula, while output from all the other implementations is always a Tarski formula by default. Both `QEPCAD B` and `RegularChains` can return Extended Tarski Formulae for QE output by definable options. In the latter case this is by passing `'output' = 'rootof'`, and the formulae produced are actually of relations on real algebraic functions using real indexing in the same way as `QuantifierElimination`. However, unlike `QuantifierElimination` `RegularChains` actually uses radicals over RootOfs when the polynomial for the RootOf is quadratic in $\_Z$. Real indexed RootOfs via `'index' = 'real[i]'` differ from "complex indexing" in that the indexing corresponding to the roots' positions along the real line. For interval indexing, the roots' position along the real line is clear. `QEPCAD B` offers support for generating output via ETF via an option, but by default will produce a Tarski formula, which may require addition of projection factors to the projection sets. `QuantifierElimination` CAD does not currently offer support for production of a Tarski formula where CAD is used for QE in this way. `QEPCAD B`'s Extended Tarski Formulae are represented very similarly to those of `QuantifierElimination`'s, via "real indexing" of roots of projection polynomials. ETFs can often be quite concise and meaningful, requiring fewer atoms than

an equivalent Tarski formula, but other times confusing via the density of information contained in the representation. This is also especially true of the representation of real algebraic numbers in `QuantifierElimination`, where by default the intervals can contain fractions of a very high precision.

Chapter 8

Closing, Conclusions and Further Work

8.1 Summary of Contributions

This project has contributed a new package `QuantifierElimination` for Maple that attempts to amalgamate various aspects of contemporary research in Quantifier Elimination to investigate the nuances of using them together. The focus of the project is a comprehensive investigation into a "poly-algorithm" between VTS and CAD to explore a first ever bespoke methodology for their usage together, enabling extra efficiency and features for QE. The package includes the first implementation of Virtual Term Substitution developed in collaboration with Maplesoft for Maple, the first implementation of a Lazard projection & lifting CAD in Maple, and the first implementation of a Lazard projection CAD with equational constraint optimisations in any context. This includes investigation of very recent research on curtains in a Lazard projection & lifting CAD to make projection and lifting with a single equational constraint complete. Much attention is paid to equational constraints in CAD, including Gröbner bases, pivot selection strategy, and the identification and resolution of curtains. Implementation of recent methodologies completing a single equational constraints confirm the efficacy of the methodology. In the case of multiple equational constraints, the CAD implementation is complete to the extent it knows when output cannot be proven mathematically correct and so produces an error, but attempts to avoid exit via error when it can deduce curtains are not a mathematical impediment to output or can otherwise be recovered from.

The CAD implementation offers use of a new feature, "lifting constraints" in order to become more amenable to a subset of problems featuring constraints that imply a hyperrectangle in space. A returning feature from research "Open CAD" can similarly make use of boolean structure of formulae beyond equational constraints. The work presents and discusses existing and new greedy strategies in various contexts across VTS and CAD alike. Such strategies often require non trivial usage of data structures, and some difficulties are highlighted here. These strategies are often examined in the context introduced by the work, such as the particular examination of equational constraints and usage of VTS and CAD together. The usage of Gröbner bases in conjunction with ECs for CAD aims to improve on existing monomial orderings used with a view to the behaviour of projection.

Various case studies to investigate data for usage of curtain recovery, Gröbner bases, and the poly-algorithm are explored to extend upon the benchmarking. Further features related to QE including witnesses & incrementality are explored in depth, and we have provided the first known methodology for production of witnesses when VTS and CAD are used together to achieve homogeneous QE. Witnesses are a feature involved in a greater scheme to provide rich output for users in the context of QE and similar topics. Other pedagogical functions in the subpackage `QuantifierTools` enable users to understand Tarski-like formulae further, and most functions offer a wealth of keyword options with defaults to customize usage of QE. The particular usage of RootOfs in Maple to represent real algebraic numbers and functions is developmental with respect to low level operations, but has a view to informing users with potentially more readable output. Output data structures are highly examinable, such that for example one can query properties of cells, which again often displays output in terms of real algebraic numbers or functions. A major view is taken towards intelligible interface for the package.

Incrementality is made more general in terms of "evolutionary" algorithms that additionally act at specific "atomic positions" to assist users to understand QE problems as efficiently as possible. This makes incrementality more general than is typically presented in research. The evolutionary algorithms are highly contextual to the algorithms beneath `QuantifierElimination`, and largely reconcile with an object oriented approach to the aspects of VTS and CAD and retention of data. The evolutionary methods are comprehensively presented in the contexts of VTS, the poly-algorithm, and CAD for QE. The object oriented approach continues a tree based canonicalization of VTS, enabling the poly-algorithmic methodologies and raising further exploration on the state of VTS in terms of elimination of one block of quantifiers, with its ramifications on usage of CAD to complete QE.

8.2 Conclusions

Various implementations of CAD and QE have been benchmarked against each other, also exploring the frequency of nullification & curtain occurrences in CAD, the efficacy of various variable strategies and other options in CAD, and the efficacy of `QuantifierElimination`'s QE by pure CAD against a standard VTS into CAD approach to QE or poly-algorithmic approach to QE. In terms of CAD, the benchmarking finds that the projection based variable strategies for CAD are especially effective for the Lazard projection and lifting CAD that is highly sensitive to the projection step of CAD due to equational constraints. In terms of QE, the poly-algorithm's applicability in a non trivial sense is limited, but is rarely a detriment in comparison to the "standard" approach for VTS into CAD on amenable examples. In some cases it enables extra performance, and always enables full production of witnesses and specific methodology for further general evolutionary operations where applicable due to the association of a CAD to at most one VTS node at a time. Use of VTS before CAD can be varied, often being helpful, while sometimes frustrating CAD in terms of degree, however this is likely nuanced in terms of simplification and maximum degree for elimination by VTS. VTS is largely a win on linear problems over CAD, including

those from economics, but the provided QE by CAD can be surprisingly effective on such problems as well, likely due to enumerable ECs. Curtains remain an obstruction, although case studies confirm that the methodology for "avoidance before resolution", including identification of point curtains can be a boon. In particular low level curtains prevent multiple equational constraints from reaching full potential as a complete optimisation.

8.3 Further Work

One notes that this work identifies various open problems throughout. Beyond that, there are some immediate ways to improve `QuantifierElimination` and otherwise extend upon the work of this project. Other further work is more mathematical, or otherwise outside the scope of `QuantifierElimination`.

VTS Simplification

VTS in `QuantifierElimination` suffers from only featuring "weak" rather than "strong" simplification (Section 2.4.2), which can make both intermediate and output formulae where VTS is concerned very messy. Strong simplification reflects that we should aim for candid representations of formulae, but not at cost to outweigh the benefits. `QuantifierElimination` implementing a better simplifier should immediately improve the package on the whole. A non exhaustive list of relevant works is [9, 23, 7, 40, 16]. Further, this simplification would likely affect the methodology of the poly-algorithm. While the simplification could improve the formulae formed out of multiple `IQER`s, it would also act on the individual formulae held by any one `IQER`, which may allow them to manifest meaningful truth values in lieu of usage of the poly-algorithm, or otherwise the formulae received by CAD within the poly-algorithm may become simpler. Whether this makes a better or worse case for the poly-algorithm, including in terms of the "poly-share criteria" remains to be seen.

Section 4.3 highlights that strong simplification would clearly be beneficial on the formulae held by individual `IQER`s, but can complicate matters when used to deduce the quantifier free equivalent implied by VTS at any one time, in terms of the very tree based formulations of many of the operations used in the package for VTS (in particular, tree traversals). The poly-algorithmic methodology is uncomplicated by simplification on individual `IQER`s, and may benefit from it significantly.

Tarski Formulae for CAD Output in QE

`QuantifierElimination`'s CAD currently always outputs an extended Tarski formula (Definition 33) when used to achieve QE. More specifically, this is as a result of cell descriptions always being formulae of relations on real algebraic functions (Section 3.4), hence the overall produced formula for QE output is a disjunction of such formulae (3.1). `QEPCAD B` offers support for output of an extended Tarski formulae (in `QEPCAD B`'s format), but also a Tarski formula via potential addition of polynomials to the projection

sets such that the CAD is "projection definable" [12, 8], but `QuantifierElimination` does not offer support for this at present (Section 7.5).

Negations in Universal QE in VTS

VTS in `QuantifierElimination` currently liberally distributes negations through formulae to achieve elimination of universal quantifiers, but it could be refactored such that it always treats the universal case as an existential one as to not require continual nested negations of formulae (Section 2.2).

Polynomial Operations in CAD

As per Section 3.4.1, Maple awaits some improvements to low level functions regarding real root isolation & refinement, which immediately improves the efficiency of its use beneath CAD, and minimises the risk of the associated lifting failure. This additionally applies to evaluation of relations of polynomials of real algebraic numbers for evaluation of the truth values of cells, which is another current type of lifting failure.

Data Structures for Storage

`QuantifierElimination` uses a mutable container `QEContainer` for storage of intermediate `IQER`s and `CADCell`s. It supports addition of elements, removal of elements at generic positions, and is iterable, but is rudimentary in the sense that strategy for selection of the next object to "propagate" on is $\mathcal{O}(k)$ when there are k elements in the container. Sections 2.3.1 and 3.9 highlight that other structures such as a heap may be more efficient, but one must consider the costs associated to arbitrary removal of elements as well, other mitigations such as minor restrictions on strategy, and even the context of the QE problem in terms of its quantifiers.

Further Benchmarking

`QuantifierElimination` provides a wealth of keyword options available to customize usage of QE and CAD alike. Due to the combinatorics of such, this project cannot hope to benchmark every possible combination, but further investigation of the poly-algorithm including the optimal value of the macro `POLY_SHARE_THRESHOLD` (Open Problem 70) is very canonical further work. One could even replace the "poly-share criteria" (Section 4.1) with another criterion. More canonical further work includes investigation of whether restricting VTS to merely linear elimination rather than quadratic is more efficient (i.e. varying the value of the keyword option `'MaxVSDegree'`). The "top level" evolutionary methods that act at atomic positions could be investigated, for example in terms of their usage to process an SMT formula over the theory of real arithmetic by full incrementality. Its use beneath actual SMT solving software could be interesting, especially with or against Maple's current package for SMT `SMTLIB`, which uses `Z3` [75]. Most strategies within `QuantifierElimination` are greedy, and try to aim to deduce a meaningful truth value as soon as possible, which is most oriented towards the fully existentially quantified context of `QF_NRA`. As an example, the

poly-algorithm by default proceeds with depth-wise traversal of every tree. However, further code development and refactoring to optimise the package should likely precede its presentation to SMT, where most examples will be in many more variables than those used for the benchmarking of this project. The economics examples used within the QE benchmarking (Section 7.4.4) are "SMT-like", being almost always fully existentially quantified, but only ever in low degree, hence CAD was never used beneath the poly-algorithm there. Various features such as lifting constraints and Open CAD could be benchmarked further on applicable examples via usage of `QuantifierTools`' SuggestCADOptions (Chapter 6), and the greedy CAD variable strategy could also be used, as as the better suggested default option for CAD variable strategy via the CAD benchmarking of this work. Lastly, this work does not largely benchmark against QE implementations outside of Maple, and `Redlog` [24] implemented in `REDUCE` would be another natural point of comparison, having an implementation of VTS.

`QuantifierElimination` currently does not export a function providing just VTS without usage of CAD, and if it were to do so then one could more easily explore the frequency of ineligible `IQER`s (i.e. degree violations) and hence the rate at which VTS fails to be complete, at least within the context of the implemented strategies. Such a function may return a quantified formula when ineligible `IQER`s arose without a meaningful truth value. Further, the frequency at which VTS yields at least two leaf `IQER`s, at least one of which is ineligible, is commensurate with the frequency at which the poly-algorithm inherits scope to act with different methodology than the `'whole'` methodology. The cases where VTS yields at least two ineligible `IQER`s are the cases where the poly-algorithm *may* have scope to act in an incremental way, depending on the truth value of the first `IQER` processed. These cases can better be inspected by such a function that returns a quantified formula, where the ineligible `IQER`s are inherently quantified.

Lazard Projection with Multiple Equational Constraints

While [56] provided completeness for usage of the Lazard projection with usage of a single equational constraint in projection, one notes the same completeness is yet to be provided for multiple equational constraints (Section 3.7.2). `QuantifierElimination` offers support for multiple equational constraints via the keyword argument `'UseEquations' = 'multiple'` at the risk of low level curtain errors. Even then. usage of CAD with multiple equational constraints is used under the hood in the poly-algorithm where the chance of recovery in some context is estimated to be high — often the avoidance and recovery methodologies enable completion of examples in using multiple ECs in any context, as can be seen often in the QE benchmarking. The framework for avoidance & recovery from curtains easily still applies if an algorithm (or algorithms) to decompose low level curtains is provided, but this is more of a theoretical improvement in comparison to the more practical discussions of Section 3.7.2. Development of methodologies presented here may allow CAD in any context to ignore curtains further, or otherwise identify such curtains as non obstructive point curtains — Open Problem 62 offers opportunity for further practical improvement to avoidance of curtains. The benchmarking highlights that curtains are still obstructive to varying

extents in differing contexts, giving impetus to this area of further research.

Cubic VTS

The cubic case for VTS in `QuantifierElimination` such that `'MaxVSDegree'` can be specified as 3 to enable elimination of variables appearing cubically in irreducible polynomials is under development, without cubic clustering. The cubic case was much of the contribution of [43]. Naturally, the appearance of cubic VTS within `QuantifierElimination` may affect the case for poly-algorithmic QE. A description of the current state of cubic VTS in `QuantifierElimination` may be added to the repository [65].

NuCAD and CAC in Maple

Non-uniform Cylindrical Algebraic Decomposition (NuCAD) is a technique introduced in [10, 14]. It shares many similarities with CAD. At present, it can be used as a tool for SMT over the theory of real arithmetic. As of the time of writing, there is no implementation of NuCAD for Maple in development known to the author. Such an implementation could be used to improve `QuantifierElimination`, in particular because many "SMT-like" problems about sign conditions of single polynomials arise in the course of VTS (functions such as at-cs-fac from [43] ask questions about whether the leading coefficient of a polynomial for candidate solution generation is always positive, or can vanish, etc.).

Cylindrical Algebraic Coverings (CAC) are again a variation on CAD, and were introduced with [2]. Again, the algorithm is a tool for SMT over the theory of real arithmetic, processing conjunctions of real polynomial constraints. The work states that the algorithm takes inspiration from incremental CAD, `NLSAT` [41], and NuCAD. The conflict driven search of the algorithm is essentially a concept from SAT solving. [1] highlights that in comparison to QE tools such as VTS or CAD, CAC is far more amenable to providing algorithmic proof of falsity (**UNSAT**) for a fully existentially quantified problem (hence analogously proof of truth for a fully universally quantified problem). In contrast, such a proof from VTS or CAD requires proof of completeness of VTS or CAD (that is, the confidence that the (virtual) substitutions used by either algorithm were a complete covering of all those that were necessary). Meanwhile, both VTS and CAD can prove satisfiability via witnesses (Sections 2.5 and 3.10). Again, as of the time of writing, there is no implementation of CAC for Maple in development known to the author.

Bibliography

[1] E. Ábrahám, J.H. Davenport, M. England, G. Kremer, and Z. Tonks. New Opportunities for the Formal Proof of Computational Real Geometry? In *Proceedings $\mathcal{SC}^2$ Workshop 2020*, 2020. URL: `http://ceur-ws.org/Vol-2752/paper13.pdf`.

[2] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings, 2020. URL: `https://arxiv.org/abs/2003.05633`.

[3] P. Alvandi, C. Chen, F. Lemaire, M.M. Maza, and Y. Xie. The RegularChains Library. Accessed: 18/11/2020. URL: `http://www.regularchains.org/`.

[4] T. Becker and V. Weispfenning (with H. Kredel). *Groebner Bases. A Computational Approach to Commutative Algebra*. Springer Verlag, 1993. `doi:10.1007/978-1-4612-0913-3`.

[5] R. Bradford, J.H. Davenport, M. England, and D. Wilson. Optimising problem formulation for Cylindrical Algebraic Decomposition. In *Intelligent Computer Mathematics*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-39320-4\_2`.

[6] M. Brain, J.H. Davenport, and A. Griggio. Benchmarking Solvers, SAT-style. In *Proceedings $\mathcal{SC}^2$ Workshop 2019*. CEUR Workshop Proceedings, 2017. URL: `http://ceur-ws.org/Vol-1974/RP3.pdf`.

[7] C W. Brown and A. Strzebonski. Black-Box/White-Box Simplification and Applications to Quantifier Elimination. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC 2010, pages 69–76, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1837934.1837953`.

[8] Christopher W. Brown. QEPCAD B: A Program for Computing with Semi-algebraic Sets Using CADs. *SIGSAM Bull.*, 37(4):97–108, December 2003. `doi:10.1145/968708.968710`.

[9] Christopher W. Brown. Fast Simplifications for Tarski Formulas Based on Monomial Inequalities. *J. Symb. Comput.*, 47(7):859–882, July 2012. `doi:10.1016/j.jsc.2011.12.012`.

[10] Christopher W. Brown. Open Non-uniform Cylindrical Algebraic Decompositions. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '15, pages 85–92, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2755996.2756654, doi:10.1145/2755996.2756654.

[11] Christopher W. Brown, M'hammed El Kahoui, Dominik Novotni, and Andreas Weber. Algorithmic methods for investigating equilibria in epidemic modeling. *Journal of Symbolic Computation*, 41(11):1157 – 1173, 2006. Special Issue on the Occasion of Volker Weispfenning's 60th Birthday. URL: http://www.sciencedirect.com/science/article/pii/S074771710600054X, doi:10.1016/j.jsc.2005.09.011.

[12] C.W. Brown. *Solution Formula Construction for Truth-Invariant CADs*. PhD thesis, University of Delaware, 1999.

[13] C.W. Brown. Companion to the tutorial: Cylindrical Algebraic Decomposition, ISSAC 2004, 2004. Accessed: 07/12/2020. URL: https://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/handout.pdf.

[14] C.W. Brown. Constructing a single cell in cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 70:133–140, 06 2013. doi:10.1145/2465506.2465952.

[15] C. Chauvin, M. Müller, and Andreas Weber. An Application of Quantifier Elimination to Mathematical Biology. In J. Fleischer, J. Grabmeier, F. W. Hehl, and W. Küchlin, editors, *Computer Algebra in Science and Engineering*, pages 287–296. Zentrum für Interdisziplinäre Forschung, World Scientific, August 1994.

[16] Changbo Chen and Marc Moreno Maza. Simplification of Cylindrical Algebraic Formulas. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 119–134, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24021-3_9.

[17] Changbo Chen and Marc Moreno Maza. An Incremental Algorithm for Computing Cylindrical Algebraic Decompositions. In Ruyong Feng, Wen-shin Lee, and Yosuke Sato, editors, *Computer Mathematics*, pages 199–221, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-43799-5_17.

[18] G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Proceedings 2nd. GI Conference Automata Theory & Formal Languages*, pages 134–183, 1975. doi:10.1007/3-540-07407-4_17.

[19] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24318-4_26.

[20] D.A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Undergraduate Texts in Mathematics. Springer, Heidelberg, 2015. URL: http://dx.doi.org/10.1007/978-3-319-16721-3, doi:10.1007/978-3-319-16721-3.

[21] J.H Davenport. Computer Algebra. Accessed: 30/03/2020. URL: http://staff.bath.ac.uk/masjhd/JHD-CA.pdf.

[22] J.H. Davenport and J. Heintz. Real Quantifier Elimination is Doubly Exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988. doi:10.1016/S0747-7171(88)80004-X.

[23] A. Dolzmann and T. Sturm. Simplification of Quantifier-free Formulae over Ordered Fields. *Journal of Symbolic Computation*, 24(2):209–232, 1997. URL: http://www.sciencedirect.com/science/article/pii/S0747717197901231, doi:10.1006/jsco.1997.0123.

[24] A. Dolzmann, T. Sturm, and M. Košta. Redlog — Computing with Logic. Accessed: 19/04/2019. URL: http://www.redlog.eu/.

[25] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient Projection Orders for CAD. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 111–118, New York, NY, USA, 2004. ACM. doi:10.1145/1005285.1005303.

[26] H. Du and N. Alechina. Qualitative Spatial Logic over 2D Euclidean Spaces is Not Finitely Axiomatisable. In *AAAI Conference on Artificial Intelligence 2019*, volume 33, pages 2776–2783, 07 2019. doi:10.1609/aaai.v33i01.33012776.

[27] M. England, D.J. Wilson, R. Bradford, and J.H. Davenport. Using the Regular Chains Library to build Cylindrical Algebraic Decompositions by projecting and lifting. In H. Hong and C. Yap, editors, *Mathematical Software — ICMS 2014*, pages 458–465. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-44199-2_69.

[28] Matthew England, Russell Bradford, and James H. Davenport. Cylindrical algebraic decomposition with equational constraints. *Journal of Symbolic Computation*, 100:38–71, 7 2019. doi:10.1016/j.jsc.2019.07.019.

[29] Matthew England and David Wilson. *An Implementation of Sub-CAD in Maple*. Number CSBU-2015-01 in Department of Computer Science Technical Report Series. Department of Computer Science, University of Bath, March 2015. Accessed: 18/11/2020. URL: https://researchportal.bath.ac.uk/en/publications/an-implementation-of-sub-cad-in-maple.

[30] D. Florescu and M. England. Algorithmically generating new algebraic features of polynomial systems for machine learning. In *Proceedings $\mathcal{SC}^2$ Workshop 2019*. CEUR Workshop Proceedings, 10 2019. URL: http://ceur-ws.org/Vol-2460/paper4.pdf.

[31] D. Florescu and M. England. Machine Learning to Improve Cylindrical Algebraic Decomposition in Maple. In Jürgen Gerhard and Ilias Kotsireas, editors, *Maple in Mathematics Education and Research - 3rd Maple Conference, MC 2019, Proceedings*, Communications in Computer and Information Science, pages 330–333, United Kingdom, 2020. Springer. doi:10.1007/978-3-030-41258-6_25.

[32] Patrizia Gianni. Properties of Gröbner bases under specializations. In James H. Davenport, editor, *EUROCAL '87*, pages 293–297, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51517-8_128.

[33] Redlog group. Redlog Example Management and Information System. Accessed: 26/06/2021. URL: https://www.redlog.eu/remis/.

[34] SyNRAC Group. SyNRAC QE Example Database, 2017. Accessed: 28/05/2020. URL: https://github.com/hiwane/qe_problems.

[35] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '90, pages 261–264, New York, NY, USA, 1990. ACM. URL: http://doi.acm.org/10.1145/96877.96943, doi:10.1145/96877.96943.

[36] H. Hong and G.E. Collins. Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *Journal of Symbolic Computation*, pages 299–328, 1991. doi:10.1016/S0747-7171(08)80152-6.

[37] Z. Huang, M. England, J.H. Davenport, and L. Paulson. Using Machine Learning to Decide When to Precondition Cylindrical Algebraic Decomposition With Groebner Bases. In *18th Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2017. doi:10.1109/SYNASC.2016.020.

[38] Zongyan Huang, Matthew England, David Wilson, James H. Davenport, and Lawrence C. Paulson. A Comparison of Three Heuristics to Choose the Variable Ordering for Cylindrical Algebraic Decomposition. *ACM Commun. Comput. Algebra*, 48(3/4):121–123, 2 2015. doi:10.1145/2733693.2733706.

[39] Nikolaos Ioakimidis. Sharp bounds based on quantifier elimination in truss and other applied mechanics problems with uncertain, interval forces/loads and other parameters. Technical report, 08 2019. doi:10.13140/RG.2.2.22662.52803.

[40] H. Iwane and H. Anai. Formula Simplification for Real Quantifier Elimination Using Geometric Invariance. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC 2017, pages 213–220, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087604.3087627.

[41] Dejan Jovanović. Solving Nonlinear Integer Arithmetic with MCSAT. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 330–346, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-52234-0_18.

[42] Michael Kalkbrener. Solving systems of Algebraic Equations by using Gröbner Bases. In James H. Davenport, editor, *EUROCAL '87*, pages 282–292, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51517-8_127.

[43] M. Košta. *New Concepts for Real Quantifier Elimination by Virtual Substitution*. PhD thesis, Universität des Saarlandes, 2016. doi:10.22028/D291-26679.

[44] Košta, M. and Sturm, T. and Dolzmann, A. Better answers to real questions. *Journal of Symbolic Computation*, 74:255 – 275, 5 2016. URL: http://www.sciencedirect.com/science/article/pii/S0747717115001078, doi:10.1016/j.jsc.2015.07.002.

[45] Gereon Kremer and Erika Ábrahám. Fully Incremental Cylindrical Algebraic Decomposition. *Journal of Symbolic Computation*, 100:11–37, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0747717119300847, doi:10.1016/j.jsc.2019.07.018.

[46] D. Lazard. An Improved Projection for Cylindrical Algebraic Decomposition. In Chandrajit L. Bajaj, editor, *Algebraic Geometry and its Applications*, pages 467–476. Springer New York, New York, NY, 1994. doi:10.1007/978-1-4612-2628-4_29.

[47] Maza, M. M. and Chen, C. Quantifier elimination by cylindrical algebraic decomposition based on regular chains. *Journal of Symbolic Computation*, 75:74 – 93, 2016. Special issue on the conference ISSAC 2014: Symbolic computation and computer algebra. URL: http://www.sciencedirect.com/science/article/pii/S0747717115001078, doi:10.1016/j.jsc.2015.11.008.

[48] S. McCallum. On projection in CAD-based quantifier elimination with equational constraint. In *Proceedings ISSAC 1999*, pages 145–149, 1999. doi:10.1145/309831.309892.

[49] S. McCallum. On propagation of equational constraints in CAD-based quantifier elimination. In *Proceedings ISSAC 2001*, pages 223–231, 2001. doi:10.1145/384101.384132.

[50] Scott McCallum. An improved Projection Operation For Cylindrical Algebraic Decomposition of Three-dimensional Space. *Journal of Symbolic Computation*, 5(1):141–161, 1988. doi:10.1016/S0747-7171(88)80010-5.

[51] Scott McCallum, Adam Parusiński, and Laurentiu Paunescu. Validity proof of Lazard's method for CAD construction. *Journal of Symbolic Computation*, 92:52–69, 2019. doi:10.1016/j.jsc.2017.12.002.

[52] David Monniaux. Quantifier elimination by lazy model enumeration. In Byron Cook, Paul Jackson, and Tayssir Touili, editors, *CAV 2010*, volume 6174 of *Lecture notes in computer science*, pages 585–599, Edimburgh, United Kingdom, July 2010. Springer-Verlag. URL: https://hal.archives-ouvertes.fr/hal-00472831.

[53] Casey Mulligan, Russell Bradford, James H. Davenport, Matthew England, and Zak Tonks. Dataset of automated economic reasoning problems for QE / SMT, April 2018. doi:10.5281/zenodo.1226892.

[54] C.B. Mulligan, R. Bradford, J.H. Davenport, M. England, and Z. Tonks. Nonlinear Real Arithmetic Benchmarks derived from Automated Reasoning in Economics. In *Proceedings $\mathcal{SC}^2$ Workshop FLoC 2018*. CEUR Workshop Proceedings, 2018. URL: http://ceur-ws.org/Vol-2189/paper2.pdf.

[55] A. Nair. *Curtains in Cylindrical Algebraic Decomposition*. PhD thesis, University of Bath, 2021. To appear.

[56] A. Nair, J. Davenport, and G. Sankaran. Curtains in CAD: Why Are They a Problem and How Do We Fix Them? In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *Mathematical Software — ICMS 2020*, volume 12097 of *Lecture Notes in Computer Science*, pages 17–26, Cham, 2020. Springer. doi:10.1007/978-3-030-52200-1_2.

[57] Akshar Sajive Nair, James Davenport, and Gregory Sankaran. On Benefits of Equality Constraints in Lex-Least Invariant CAD. In *Proceedings $\mathcal{SC}^2$ Workshop 2019*. CEUR Workshop Proceedings, 10 2019. URL: http://ceur-ws.org/Vol-2460/paper6.pdf.

[58] K. Röbenack, R. Voßwinkel, and H. Richter. Calculating Positive Invariant Sets: A Quantifier Elimination Approach. *Journal of Computational and Nonlinear Dynamics*, 14:1–5, 2019. doi:10.1115/1.4043380.

[59] A. Seidl. *Cylindrical Decomposition Under Application-Oriented Paradigms*. PhD thesis, Universität Passau, 2006. URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/year/2006/docId/46, doi:10.22028/D291-26679.

[60] D. Stoutemyer. Ten commandments for good default expression simplification. *J. Symbolic Comp.*, 46:859–887, 2011. doi:10.1016/j.jsc.2010.08.017.

[61] Adam Strzebonski. Real Polynomial Systems, Wolfram Mathematica. URL: https://reference.wolfram.com/language/tutorial/RealPolynomialSystems.html.

[62] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. 2nd ed., Univ. Cal. Press. Reprinted in *Quantifier Elimination and Cylindrical Algebraic Decomposition* (ed. B.F. Caviness & J.R. Johnson), Springer-Verlag, Wein-New York, 1998, pp. 24–84., 1951. doi:10.1007/978-3-7091-9459-1_3.

[63] Z. Tonks. Evolutionary Virtual Term Substitution in a Quantifier Elimination System. In *Proceedings $\mathcal{SC}^2$ Workshop 2019*. CEUR Workshop Proceedings, 10 2019. URL: http://ceur-ws.org/Vol-2460/paper7.pdf.

[64] Z. Tonks. A Poly-algorithmic Quantifier Elimination Package in Maple. In Jürgen Gerhard and Ilias Kotsireas, editors, *Maple in Mathematics Education and Research*, pages 171–186, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-41258-6_13.

[65] Z. Tonks. Repository of data supporting the thesis "Poly-algorithmic Techniques in Real Quantifier Elimination", 2020. Accessed: 04/01/2021. URL: https://zenodo.org/record/4382083, doi:10.5281/zenodo.4382083.

[66] Z. Tonks. VTS and Lazard Projection CAD in Quantifier Elimination with Maple. Technische Universität Braunschweig, 2020. Last accessed: 06/08/2020. doi:10.5446/48017.

[67] Tonks, Z. Quantifier Elimination and projection & lifting Cylindrical Algebraic Decompositions in the QuantifierElimination Package in Maple, 2020. Video of Software Demo recorded for Maplesoft Conference 2020. Accessed: 16/11/2020. URL: https://www.youtube.com/watch?v=n4u0_IxWCUc.

[68] Yumi Wada, Takuya Matsuzaki, Akira Terui, and Noriko H. Arai. An Automated Deduction and Its Implementation for Solving Problem of Sequence at University Entrance Examination. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software — ICMS 2016*, pages 82–89, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-42432-3_11.

[69] V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1):3–27, 1988. doi:10.1016/S0747-7171(88)80003-8.

[70] V. Weispfenning. Quantifier Elimination for Real Algebra — the Quadratic Case and Beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1 1997. doi:10.1007/s002000050055.

[71] D. Wilson, J. H. Davenport, M. England, and R. Bradford. A "Piano Movers" Problem Reformulated. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 53–60, Sep. 2013. doi:10.1109/SYNASC.2013.14.

[72] David Wilson. Real Geometry and Connectedness via Triangular Description: CAD Example Bank, April 2013. Accessed: 18/11/2020. URL: https://researchdata.bath.ac.uk/69/.

[73] David Wilson, Russell Bradford, and James Davenport. Speeding up Cylindrical Algebraic Decomposition by Gröbner Bases. In *CICM 2012*, pages 279–293. Springer LNCS 7362, 2012. doi:10.1007/978-3-642-31374-5_19.

[74] Hitoshi Yanami and Hirokazu Anai. SyNRAC: A Maple Toolbox for Solving Real Algebraic Constraints. *ACM Commun. Comput. Algebra*, 41(3):112–113, September 2007. doi:10.1145/1358190.1358205.

[75] Z3 Group. The Z3 Theorem Prover. Accessed: 26/10/2020. URL: https://github.com/Z3Prover/z3.

[76] G. M. Ziegler. Sharir's Cube, 2000. Accessed: 02/12/2019. URL: http://www.eg-models.de/models/Polytopes/2000.09.028/_preview.html.