aa: Appendix

Gordon Krieger
gordon.krieger@mail.mcgill.ca

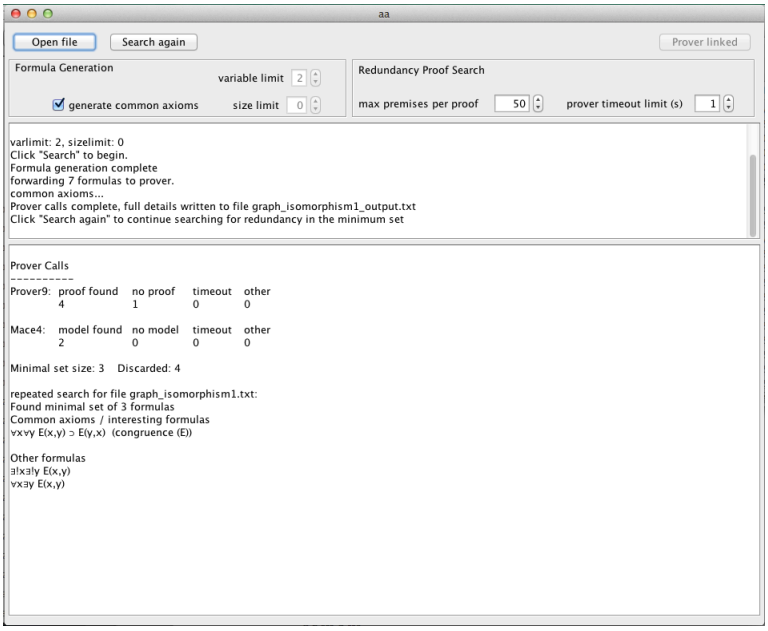
July 28, 2019

0 Installation

Source code, documentation, input files and an executable for the application are hosted at `https://github.com/gsfk/aa`.

For most purposes, it is sufficient to run the application from the executable .jar file. This is tested for OSX only. Operation requires an installation of the theorem prover Prover9 and model checker Mace4, downloadable together from `https://www.cs.unm.edu/~mccune/mace4/download/`. On first running the program `aa` needs to know where your copy of Prover9 and Mace4 are, they should be in the folder `LADR-2009-11A/bin/` from your Prover9/Mace4 download. Press the "Link Prover" button to connect to your Prover9 executable.

1 GUI



The application has two main windows: the upper window is for status and error messages, while the lower window shows statistics for prover and model checker calls, as well as the final search results.

General Controls:

Link Prover
Spawns an open file dialog so you can direct the software to your copy of Prover9 and Mace4.

Open File
Opens a file.

Search
Perform an initial search for axioms for the input domain. This initiates a round of formula generation and theorem prover calls. To perform a search on the same file but with different formula generation settings, re-open the file and change the settings accordingly. On starting search, a progress bar will pop-up in front of the main window, showing progress of formula generation, theorem prover / model checker calls, and allowing the user to cancel search.

Search Again
After the first search on any file, the "Search" button changes to read "Search again": this performs an additional search for redundancy through the minimum set returned. Additional repeat searches can be performed an arbitrary number of times; proof search settings can be changed each time.

Controls for search parameters:

1: Formula Generation

"generate common axioms": include common mathematical axioms such as commutativity or transitivity in the search. See section 6, "Extended search" below.

variable limit: the maximum number of variables to use in generated formulas (maximum value of 4)

size limit: the maximum number of logical operators to use in generated formulas (zero or greater). Both the variable limit and size limit have an exponential effect on the number of formulas generated, so these values should be modest.

2: Redundancy proof search

max premises per proof: The maximum number of premises to use in each proof. See section 5.1 below.

prover timeout limit: the maximum amount of time in seconds to wait for a result from the theorem prover and model checker. (1 second or greater)

Full details of each search are written to an output file, with the name `<inputfilename>_output.txt`.

2 Input

Input is through a text file listing all elements, relations and facts about the input. All input should represent a finite structure, since facts are listed exhaustively. For arithmetic-like structures that can be represented in table format, this means flattening the table into a list of tuples. Other structures require different approaches.

Input can be typed or untyped. For typed input, elements are divided into types, and relations can bear typing constraints. Example input files for different structures are shown below.

The group of order two (file ***g2.txt****):*

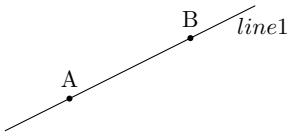
$*$	0	1
0	0	1
1	1	0

```
Elements:  0, 1

Relations:
*

Facts:
*(0,0,0)
*(0,1,1)
*(1,0,1)
*(1,1,0)
```

*A simple untyped finite geometry example: (****lines.txt****):*

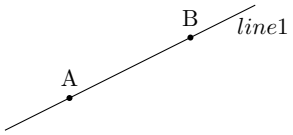



```
Elements:  A, B, line1

Relations:
isPoint
isLine
On

Facts:
isPoint(A)
isPoint(B)
isLine(line1)
On(A, line1)
On(B, line1)
```

*A typed version of the same example (****linestyped.txt****)*

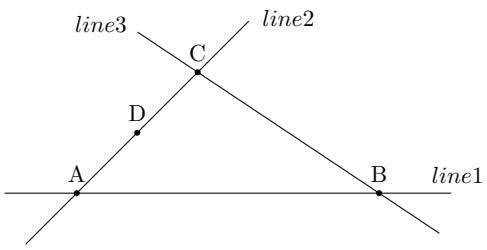



```
Point:  A, B
Line:  line1

Relations:
On(Point, Line)

Facts:
On(A, line1)
On(B, line1)
```

A slightly more complex geometry example:




```
Point:  A, B, C, D
Line:  line1, line2, line3

Relations:
On(Point, Line)
Between(Point, Point, Point)
Determines(Point, Point, Line)
Intersect(Line, Line, Point)


Facts:
On(A, line1)
On(A, line2)
On(B, line1)
On(B, line3)
On(C, line2)
On(C, line3)
On(D, line2)
Between(D, A, C)
Between(D, C, A)
Determines(A, B, line1)
Determines(B, A, line1)
Determines(A, C, line2)
Determines(C, A, line2)
Determines(C, B, line3)
Determines(B, C, line3)
Intersects(line1, line2, A)
Intersects(line2, line1, A)
Intersects(line1, line3, B)
Intersects(line3, line1, B)
Intersects(line2, line3, C)
Intersects(line3, line2, C)
```

For broader constraints, the underscore character "_" can be used as a wildcard in type constraints, for example:

```
Types:
Dog:  Fido, Rover
Cat:  Felix, Fluffy

Relations
hatesDog(_, Dog)
```

So both `hatesDog(Fido, Rover)` and `hatesDog(Felix, Rover)` are well-typed. Relations with no type constraints do not need the wildcard character:

```
Relations:
isCute
isBiggerThan
```

These are equivalent to the more verbose `isCute(_)` and `isBiggerThan(_, _)`.

There is no need to enter redundant facts about the types of each element, such as

```
Point(A)
Point(B)
Line(l1)
```

in fact this will throw an error.

All data in the input file is stored in Java object called `Input`. Search space parameters such as the maximum number of operators in a formula are stored as integers. Domain elements are stored as an ArrayList of strings. Data for each relation in the input is packaged into a `Relation` object.

`Relation` stores:

- the name and arity of the relation
- a 2D ArrayList of all the facts for this relation.

The fact table is two dimensional, since each individual fact tuple is stored as an ArrayList: the fact `On(A, line1)` above is stored as $<$ `A, line1` $>$ in the Relation object for "On". So the entire fact table for On is an ArrayList of ArrayLists:

$$<<\texttt{A, line1}>,<\texttt{B, line1}>>$$

The `Input` object holds a collection of all the `Relation` object in the input, this too is stored as an ArrayList.

3 Expression trees for logical formulas

Logical formulas are represented in the code as expression trees. The implementation is straightforward: the `FormulaTree` class defines binary trees with nodes for:

Quantifiers: $\forall x$ $\exists x$ $\exists! x$

Operators: $\&$ $\vee$ $\supset$ $\equiv$ $\neg$

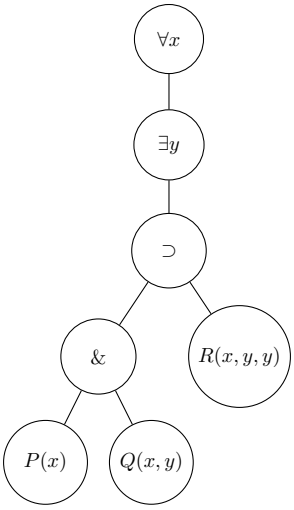
Predicates: $P(x)$ $P(y)$ $Q(x,x)$ $Q(x,y)$ $Q(y,x)$ $Q(y,y)$ ...

There are separate `Quantifier`, `Operator` and `Predicate` classes, all of them subclasses of an abstract `Node` class. There are further subclasses for particular quantifiers and logical operators.

An example formula

$$\forall x \exists y \ (P(x) \ \& \ Q(x,y)) \supset R(x,y,y)$$

is represented by the tree



where the structure of the tree reflects both quantifier scope and the structure of parentheses in the formula.

The truth value of formulas generated is determined by iterating over their corresponding expression trees and checking all claims against the input domain.

4 Formula generation

A brief digression on an earlier version of the code is called for here, since it makes the overall approach more clear. The earlier version:

1. exhaustively generates all formulas,
2. exhaustively checks the truth values of all the formulas generated,
3. sends only the true formulas to the prover to find a minimal set of formulas.

Step one involves iteratively generating formulas with increasing size, where size is measured by the number of logical operators in the formula. In general, the approach is to generate unquantified formulas first, then attach quantifiers at the front, always in a fixed x, y, z, w order. So for an example unquantified formula over two variables:

$$isPoint(x) \supset On(x, y)$$

and three quantifiers $\forall, \exists!, \exists$, there are nine possible prefixes of quantifiers:

$$\begin{array}{ccc} \forall x \forall y & \forall x \exists! y & \forall x \exists y \\ \exists! x \forall y & \exists! x \exists! y & \exists! x \exists y \\ \exists x \forall y & \exists x \exists! y & \exists x \exists y \end{array}$$

In older versions of the code, all possible quantifier prefixes were added to each unquantified formula produced, so each unquantified formula over n variables would produce 3^n quantified formulas.

The current version of the code uses a redundancy elimination scheme that combines steps 1 and 2: as formulas are generated, their truth value is checked immediately. Based on the result, some redundant formulas may not be generated at all, resulting in a smaller set of true formulas to send to the prover. Since prover calls are expensive, this can represent a significant time savings.

The redundancy elimination scheme is referred to as "hand elimination" in the code, since some redundant formulas are removed "by hand" (*ie*: without relying on the prover). This is described in section 4.1 below.

Formula generation begins with the `FormulaGenerator` method `Search()`

> `public Map<String, FormulaTree> Search()`
>
> input: none (implicit parameter is a FormulaGenerator object, which includes a reference to the input)
>
> output: A set of true formulas describing the input

`Search()` calls the static method `generatePredicates()`

> `generatePredicates(ArrayList<Relation> relations, int numVars)`
>
> input: a collection of relations from the input file, and `numVars`, the search limit on the number of variables to use (1-4)
>
> output is a collection of all `Predicate` objects needed to produce all possible formulas over the given number of variables.

In the example, input is three Relation objects representing $isPoint$, $isLine$ and On, while the variable limit is two. This outputs the collection of predicate objects

$isPoint(x)$	$isLine(y)$	$On(y, x)$
$isPoint(y)$	$On(x, x)$	$On(y, y)$
$isLine(x)$	$On(x, y)$	

The "hand elimination" step produces five quantified formulas (only true formulas are returned, with some redundant formulas eliminated)

$\exists x \exists! y\ On(x, y)$	$\exists! x \exists y\ On(y, x)$	$\exists! x\ isLine(x)$
$\exists x \exists y\ On(x, y)$	$\exists x\ isPoint(x)$	

Unquantified formulas with a single operator are produced by calling `generateSubtrees()`. These are referred to as subtrees for their role later on in the code in growing smaller terms into larger ones.

> `public static ArrayList<FormulaTree> generateSubtrees (ArrayList<Predicate> predicateList)`
>
> input: the collection of predicate objects produced above
>
> output: a collection of all unquantified formulas with a single operator.

The eight predicate terms in the example produce 148 single-operator formulas. This is slightly less than the maximum possible number of formulas on this input, which is 264:

$$\begin{aligned} & 8 \text{ predicate terms} \times 4 \text{ binary operators} \times 8 \text{ predicate terms} \\ + \ & 8 \text{ predicate terms} \times 1 \text{ unary operator} \\ = \ & 264 \end{aligned}$$

The savings is from:

- not producing redundant formulas such as A & A, $A \vee A$,

- not producing redundant formulas such as B & A when A & B already exists (similarly for $\vee$ and $\equiv$),

- not producing tautological formulas such as $A \supset A$ or $A \equiv A$.

`handEliminationSearch()` on this collection produces 179 size one terms.

Again only true terms are produced, with some redundancies eliminated beforehand. Note that at this stage, many claims will be uninteresting or trivially true.

Higher size terms are produced by:

> `public static Map<String, FormulaTree> higherSizeTerms(Collection<FormulaTree> sizeMinusOneTerms, ArrayList<FormulaTree> subtrees)`
>
> input:
> `sizeMinusOneTerms`, a collection of unquantified terms of size $s - 1$
> `subtrees`, the subtree collection produced above
>
> output: a hash map of all unquantified formulas of size s

This method is called once for each formula size of two or higher, with the `sizeMinusOneTerms` parameter updated accordingly each time.

4.1 Removing redundant quantifier prefixes ("hand elimination")

This stage of the code combines a single unquantified formula with a set of quantifier prefixes, producing a set of well-formed formulas. Older versions of the code worked in this manner:

Generate an unquantified formula over n variables.

↓

Add all possible quantifier prefixes (makes 3^n formulas).

↓

Check all truth values.

↓

Return all true formulas.

The main inefficiency here is exhaustively adding all quantifier prefixes without bothering to check truth values along the way, guaranteeing an exponential blow-up in the number of formulas each time. If we instead alternate between adding prefixes and checking truth values, we might be able to discard some prefixes and produce a smaller set of formulas. This is motivated by this observation:

given an unquantified formula $\phi(x)$ and a domain of two or more elements,

$$\forall x\ \phi(x) \text{ true } \Rightarrow \begin{cases} \exists x\ \phi(x) \text{ true} \\ \exists! x\ \phi(x) \text{ false} \end{cases}$$

$$\exists! x\ \phi(x) \text{ true } \Rightarrow \begin{cases} \exists x\ \phi(x) \text{ true} \\ \forall x\ \phi(x) \text{ false} \end{cases}$$

So if for example $\forall x\ \phi(x)$ is true, we don't even need to consider the other claims: we don't need to check their truth values, and we don't need to build expression trees for them. So, assuming we checked the prefix $\forall x$ first, we can discard the prefixes $\exists! x$ and $\exists x$ unexamined.

If we consider only $\forall$ and $\exists$, then the case for multiple quantifiers is similar, *eg*:

$$\forall x \exists y \forall z\ \phi(x,y,z) \text{ true} \Rightarrow \begin{cases} \forall x \exists y \exists z\ \phi(x,y,z) \\ \exists x \exists y \forall z\ \phi(x,y,z) \\ \exists x \exists y \exists z\ \phi(x,y,z) \end{cases} \text{ true.}$$

Any number of universal quantifiers can be swapped for existential ones to generate a weaker true claim. Trouble starts to loom when we extend this technique to uniqueness quantification. Since $\exists! x\ \phi(x) \Rightarrow \exists x\ \phi(x)$, can we swap any uniqueness quantifier for an existential one and always expect to get a redundant claim? No! Worse, even the simple trick of replacing $\forall$ with $\exists$ breaks down when a uniqueness quantifier is present.

Here is a simple example, using the "ancestor" relation $anc(x, y)$, read as "x is the ancestor of y". Each integer is its own ancestor, and the ancestor of any numbers greater than it. Given elements $1, 2, 3$, and facts

```
anc(1,1)
anc(1,2)
anc(1,3)
anc(2,2)
anc(2,3)
anc(3,3)
```

this uniqueness claim is true:

$\exists! x \exists! y\ anc(x, y)$ (there is a unique element that is the ancestor of only one element, *ie*: element 3).

But swapping the inner $\exists!$ for $\exists$ produces a false claim:

$\exists! x \exists y\ anc(x, y)$ (only one element is an ancestor).

And replacing $\forall$ with $\exists$ breaks down here as well, since this claim is true:

$\exists! x \forall y\ anc(x, y)$ (there is a unique element that is the ancestor of all elements)

but changing the $\forall$ to $\exists$ produces the same false claim above.

The one time we can swap a uniqueness quantifier for an existential one and always get a correct, redundant claim is when we are changing the outermost quantifier. So, recalling the true claim $\exists! x \exists! y\ anc(x, y)$ above, this claim is also true:

$\exists x \exists! y\ anc(x, y)$ (there is an element that is the ancestor of only one element).

This applies generally, since $\exists!\phi(x) \Rightarrow \exists\phi(x)$ holds, even if $\phi(x)$ is a term that happens to include quantifiers for variables other than x.

The starting point in the code for hand elimination is `handEliminationSearch()`:

> `public ArrayList<FormulaTree> handEliminationSearch(FormulaTree term)`
>
> input: `term`, the expression tree for a single unquantified formula, as well as the implicit parameter, a `FormulaGenerator` object.
>
> output: a collection of true quantified formulas, all of the form "quantifier prefix + term", possibly with some redundant formulas absent.

So the new workflow looks like this:

Generate an unquantified formula ϕ over n variables.

$\downarrow$

Generate the set of 3^n quantifier prefixes.

$\downarrow$

Iterate through all prefixes, testing the truth value of the formula "prefix + ϕ".
If true, remove any prefixes from the set that would lead to redundant formulas.

$\downarrow$

Return all true formulas generated.

Returning to $isPoint(x) \supset On(x, y)$, this has two variables, so the set of possible quantifier prefixes (in the order they are considered) is:

$\forall x \forall y$
$\forall x \exists! y$
$\forall x \exists y$
$\exists! x \forall y$
$\exists! x \exists! y$
$\exists! x \exists y$
$\exists x \forall y$
$\exists x \exists! y$
$\exists x \exists y$

Formulas are constructed and tested until we find a true one, which begins a search for redundancies. The first two formulas constructed are false. The third formula $\forall x \exists y\ isPoint(x) \supset On(x, y)$ is true. This initiates a call to `redundantPrefixes()` to return prefixes redundant to $\forall x \exists y$: this produces the single prefix $\exists x \exists y$, which is removed from the set.

The next formula is $\exists!x \forall y\ isPoint(x) \supset On(x, y)$ which again is true (it's true for all y only when $x = line1$). Again the call to `redundantPrefixes()` returns a single redundant prefix, this time $\exists x \forall y$, which is removed from the set. (Recall that the only situation where we currently swap $\exists!$ for $\exists$ is when the uniqueness quantifier is outermost.)

So far, two prefixes led to false claims, two to true ones, and two were removed from the set. The remaining prefixes are:

$\exists!x \exists!y$
$\exists!x \exists y$
$\exists x \exists!y$

Only the last prefix leads to a true claim. But since it contains a uniqueness quantifier not in the leftmost position, the current iteration of `redundantPrefixes()` returns nothing when asked for prefixes redundant to $\exists x \exists!y$. (In this particular case, this turns out to be an excess of caution, since $\exists x \exists y$ is in fact redundant to $\exists x \exists!y$. However this prefix has already been removed from the set.)

The end result is two true formulas returned, and two true but redundant formulas discarded, cutting in half the number of formulas sent to the prover with the particular form "$isPoint(x) \supset On(x, y)$."

Even with the current limited scope of hand elimination, there is a good reduction in the number of formulas sent to the prover. Here are statistics for two small examples. The first is `anc.txt`, the structure discussed above. The second, `big.txt`, is an input file meant to produce a large number of formulas. "Formula Size" refers to the count of logical operators in a formula, "number of formulas generated" means the number of *true* formulas that are passed along to the prover, which is equal also to the number of prover calls that need to be made.

Table 1: results for anc.txt

Formula size	Number of unquantified formulas	Number of formulas generated (with hand elimination)	Number of formulas generated (no hand elimination)
0	4	7	14
1	34	66	154
2	914	1750	4100
3	29,272	55,931	131,358
4	1,038,068	(out of memory)	(out of memory)

Table 2: results for big.txt

Formula size	Number of unquantified formulas	Number of formulas generated (with hand elimination)	Number of formulas generated (no hand elimination)
0	8	5	7
1	148	200	370
2	8724	12,013	23,316
3	602,192	850,231	(out of memory)

4.2 Triviality testing

The final step in formula generation checks all formulas generated for triviality. Although the rules for formula construction guarantee that no formula of the form $A \supset A$ will be generated for *atomic* A, larger formulas taking this shape are possible, *e.g.:*

$$(P(x) \ \& \ \neg Q(x,y)) \supset (P(x) \ \& \ \neg Q(x,y)).$$

These sorts of shapes are more difficult to avoid with simple construction rules, so instead are detected and eliminated after the fact. Formulas with trivial forms are eliminated, including those that have trivialities only as substructures. The class `TrivialityChecker` handles this task; it eliminates formulas with the following forms (A, B may be atomic or not, and "always false" for a term $P(x)$ means that $\forall x \neg P(x)$ is true):

Implications

$A \supset B$ where A is always false or B is always true

$A \supset A$

$A \supset (B \supset A)$

Biconditional

$A \equiv A$

Disjunction

$A \vee B$ where one of A or B is always false

$A \vee \neg A$, $\neg A \vee A$,

4.3 Typed Formula Generation

Several of the formula generation steps are unchanged from the original untyped version.

– produce all predicate terms, *eg*: $On(x,x), On(x,y), On(y,x), On(y,y)$

– produce all one-operator subtrees

– to grow larger terms: take an unquantified formula of size $s-1$, swap one of its predicates with a subtree, producing a formula of size s.

The differences come when adding quantifiers, which in this case are typed. For each (valid) unquantified formula f:

– generate the set of all possible typed quantifier prefixes for f (consider only the number of variables, ignore type constraints for the moment),

– produce a set of quantified formulas by making all "prefix + f" combinations (note that there is no "hand elimination" step for typed formulas),

– Typecheck all formulas. This is a simple matter of iterating through their expression trees: at the quantifiers the variables are assigned types, at the predicates they are checked against the type constraints. Badly typed formulas are discarded,

– Translate the typed formula into an untyped one:

$$(\forall x : T)\ \phi(x) \rightarrow \forall x(T(x) \supset \phi(x))$$

$$(\exists! x : T)\ \phi(x) \rightarrow \exists! x(T(x)\ \&\ \phi(x))$$

$$(\exists x : T)\ \phi(x) \rightarrow \exists x(T(x)\ \&\ \phi(x))$$

– Check truth values, discarding false formulas. Forward everything else to the prover.

The untyped formulas produced no longer present an exhaustive search of all possible formulas. Instead they come in a form typically expected of axioms: a universally quantified implication, or a (unique or not) existentially quantified conjunction. This does not mean that claims with disjunctions or biconditionals cannot be made, but that they are made in the context of a larger formula, *e.g.:*

$$\exists x Point(x)\ \&\ \exists y\ (Point(y)\ \&\ \exists z\ (Point(z)\ \&\ (Between(z,x,y) \vee Between(z,y,x))))$$

The only clear omission here is that unadorned existence claims for each type will never be made, for example

$$\exists! x\ Line(x).$$

So these are generated individually for each type in a separate step.

Typed generation works well with typed input, significantly reducing the search space of formulas by eliminating all mis-typed formulas. But on input where there are no typed constraints, the untyped approach is superior. Applying typing to an input of this sort increases the complexity of the formulas generated with no benefit, since formulas cannot be incorrectly typed.

For example, the usual representation of the closure property requires no operators at all:

$$\forall x \forall y \exists z *(x,y,z). \tag{1}$$

If we create a single type for all elements, the equivalent typed claim is

$$(\forall x : T)\ (\forall y : T)\ (\exists z : T) *(x,y,z).$$

This is ultimately translated to an untyped expression before being sent to the prover, using a new predicate $T(x)$, read as "x is a member of type T." The resulting formula has three instances of the predicate, and so three extra operators:

$$\forall x\ (T(x) \supset \forall y\ (T(y) \supset \exists z\ (T(z)\ \&\ *(x,y,z)))).$$

This is significantly more complex than (1). Input files `g2.txt` and `g2typed.txt` show this effect. The first file is untyped, and `aa` returns an axiom set of 4 formulas in less than 10 seconds. The second file is equivalent to the first one, except that all elements are gathered into a single type. The result is a minimal set of 33 formulas, even when very long prover timeout limits are used.

But for inputs where there is a natural division into types, typed expressions generally do not add to formula complexity. To make the claim "all points are on some line," the untyped approach has to rely on relations to make claims about the properties of an element:

$$\forall x \exists y\ (isPoint(x) \supset (isLine(y)\ \&\ On(x,y)). \tag{2}$$

The equivalent typed formula has no operators at all:

$$(\forall x : Point)\ (\exists y : Line)\ On(x, y)$$

and the untyped translation is just (2) again, with only slight syntactic differences:

$$\forall x\ Point(x) \supset \exists y\ (Line(y)\ \&\ On(x, y)).$$

5 Prover calls

Calls to the prover are handled by the class `ParallelProver`. Given a set T of true formulas, consider each formula f in turn, and try to prove it from the rest of the set. If you can find a proof, throw that formula away. Whatever is left is the minimal set:

1. `for each formula` f `in` T`:`
2. `if` $\{T - f\} \vdash f,\ T = \{T - f\}$
3. `return` T

This is an imperfect approach, but works well in practice, and is clearly better than the brute force alternative of generating all $2^{|T|}$ possible subsets and picking the smallest one sufficient to be an axiom set.

Where $\{T - f\} \nvdash f$, the theorem prover can search indefinitely for a proof, especially for large T, leading to a long execution time. An obvious speedup is suggested in the manual for Mace4, a model checker / model builder that accompanies the prover software Prover9:

> One of the basic ways to use Mace4 is as a complement to the theorem prover, with the prover searching for a proof, and Mace4 looking for a counterexample.

So we can spend the time spent waiting for a result from the theorem prover to look for a counterexample model, which would demonstrate that no proof is possible.

Conveniently, both Prover9 and Mace4 accept input in the same format. In our case we are listing a set of premises $\{f_1, f_2, \ldots f_{n-1}\}$, and a single formula f_n that we want to prove from the premises. Prover9 handles the proof search, while Mace4 looks for a counterexample model: it tries to construct a model for the set $\{f_1, f_2, ...f_{n-1}, \neg f_n\}$ (here we mean "model" in the logical sense, *i.e.* a structure that satisfies the set of claims being made). This last set is consistent when f_n does *not* follow from the other formulas, so there exists some model that Mace4 can find; typically the model is small enough that Mace4 can find it quickly, much faster than the time needed for the theorem prover to timeout or exhaust its search. The converse is also true; if a proof exists, typically Prover9 will find it before Mace4 gives up searching.

Both Prover9 and Mace4 are run at the same time on the same input file. They are launched as separate processes, and their creation, destruction and return values are handled by separate Java `Runnable` objects that run in distinct threads. In general we accept the opinion of whichever application finishes first, unless the first result is a timeout, then we wait for a second opinion. .

The `ParallelProver` class extends SwingWorker, Java's abstract library class for performing time-consuming background methods without freezing a graphic user interface. Time-consuming tasks are performed in the `doInBackground()` method:

`public Map<String, FormulaTree> doInBackground()`

input: none (implicit parameter is a `ParallelProver` object)

output: a minimal set of formulas (as a HashMap).

This method iterates through all the formulas passed on by the formula generation step. For each formula, it tries to prove it from the remaining formulas, creating the appropriate Prover9/Mace4 input file. After creating the file, it calls `parallelProofSearch()` to get results.

`public ProverSearchResults parallelProofSearch()`

input: implicit parameter parallelProofSearch object

output: a value from the `ProverSearchResults` enum list. The return options are:

PROOF_FOUND
NO_PROOF_FOUND
PROVER_TIMEOUT
PROVER_OTHER_RESULT
PROVER_INPUT_ERROR
PROVER_OTHER_ERROR
MACE4_COUNTEREXAMPLE_FOUND
MACE4_NO_MODELS_FOUND
MACE4_TIMEOUT
MACE4_OTHER_RESULT

`parallelProofSearch()` spawns two separate Java threads, which in turn call the OS to create two separate processes, one for Prover9 and another for Mace4. In most cases we accept the result of the first of the two to finish. The most interesting and most common results will be Prover9 finding a proof (PROOF_FOUND) and Mace4 finding a counterexample (MACE4_COUNTEREXAMPLE_FOUND). If a proof of some formula is found, the formula is redundant so not included in a minimal set; if a counterexample is found, the formula is *not* redundant, so is included.

Other cases:

NO_PROOF_FOUND: this means exhaustive search with no proof found, this is possible with smaller input files, although typically Mace4 will find a counterexample first. No proof exists, so add to minimal set.

PROVER_TIMEOUT, MACE4_TIMEOUT: these are only possible if both applications time-out. This gives us no information, so conservatively assume formula is not redundant and add to minimal set. This happens more with a larger search space.

PROVER_OTHER_RESULT: other Prover9 values, currently unused.

PROVER_INPUT_ERROR: bad input file, stop and warn user.

PROVER_OTHER_ERROR: prover crashed or some other fatal result, stop and warn user.

MACE4_NO_MODELS_FOUND: Mace4 exhausted search with no model found, possible with smaller files. This is a rare case, normally we expect Prover9 to find a proof first.

MACE4_OTHER_RESULT: other return values, typically an error.

5.1 Partial Search

A broad search space can lead to a large number of true formulas to handle. Beyond a few hundred formulas, Prover9 and Mace4 can take a noticeably long time to execute on some formulas, beyond 5 or 10 seconds. The timeout can be set arbitrarily long, but this will add significantly to the overall search time.

The goal of hand elimination (Section 4.1) was to reduce the number of formulas to work with. Another option for speedup is to relax for a moment the idea that we have to do *exhaustive* search. At each step in the prover calls stage we are trying check a formula for redundancy against *all* the other formulas. An obvious option is to check against only some subset of formulas, performing a partial search for redundancy. This sounds like an ugly compromise between speed and accuracy, but note two things: (1) we are not limited to a working with fixed subset of formulas, we just check particular formulas against a "moving window" of its neighbour formulas; (2) we can still use this method to perform exhaustive search, by the "search again" feature where we can send the minimal set back to the prover for additional redundancy checks. Under ideal circumstances, repeated search will converge to a small set of axioms. For very large sets even repeated search will sometimes fail to converge to anything useful.

Partial search is handled largely by the `PropertyChangeListener` feature of the GUI, which alters the control flow of the program based on the activity of the background threads. When the usual first round of axiom search is complete, it resets for a repeat search, feeding the output of one round of prover calls into the input of the next round.

5.2 Prover order

The prover call stage considers formulas in the reverse order that they were generated. So the largest formulas are the first to be considered for elimination, then progressively smaller formulas, and ending with any common axioms. A few experiments with prover order demonstrate that: (1) the axioms set produced depends on prover order but the variation in size between sets is typically small, and (2) although repeated rounds of prover calls can converge on similar sizes, they do not necessary converge on the same formulas.

5.3 Uniqueness quantification

Prover9 and Mace4 do not accept uniqueness quantification in their input, nor does this appear to be a feature of any other theorem prover. Uniqueness quantifiers are not included in TPTP, a standard input syntax used by multiple theorem provers. So any formula bearing a uniqueness quantifier needs to be modified before being sent to the prover. This is accomplished through formula tree rewriting.

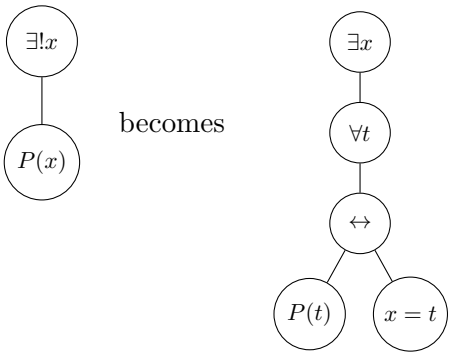
There are multiple different ways to translate formulas with uniqueness quantifiers into formulas without them. The first iteration of the project used the equivalence

$$\exists! x P(x) \iff \exists x (P(x) \wedge \forall t (P(t) \rightarrow x = t)),$$

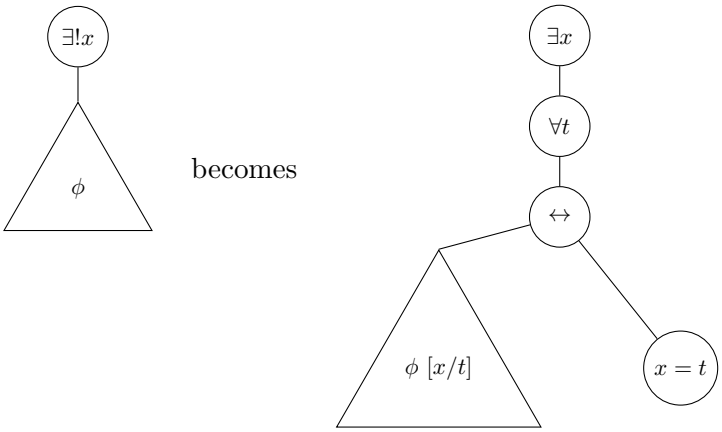
this implementation uses the slightly shorter

$$\exists! x P(x) \iff \exists x \forall t (P(t) \leftrightarrow x = t)). \tag{3}$$

The modification is accomplished by a recursive rewriting of formula trees. A copy of the original formula tree is constructed node-by-node, and in all but one case nodes are copied unchanged. When a uniqueness quantifier is encountered, the tree is modified according to the equivalence (3) above:



and in general, where ϕ is any subformula:



where a recursive call continues with the subtree ϕ, with all instances of x replaced with t.

The new formulas generated are seen only by the prover, they are not output to the user, who sees only the equivalent original formula bearing a uniqueness quantifier. In particular the input domain need not have an explicit equality relation, what is being captured is the equality implicit in any uniqueness claim.

Trees are rewritten by the recursive method `elimUniquenessRecurse()` which operates as described.

6 Extended search

The main advantage to a brute-force approach to axiom search is completeness, in the sense that any axiom existing within the defined search space is guaranteed to be found. The main drawback is the astronomical size of even the most limited search using a small number of variables or operators.

This difficulty means that many common mathematical axioms that we might expect to see lie

outside of the search space, since many of them require multiple variables, multiple operators, multiple relations, or some combination of the three. An obvious augmentation is to add known common axioms to the list of formulas to investigate.

There are two complications to consider. First, common axioms need to be produced in either typed or untyped versions, depending on the input. The second, more difficult problem is a matter of notation: the implementation uses predicate notation for all relations. If for example we want to say that x and y sum to z, the usual notation:

$$x + y = z$$

becomes, for example

$$Add(x, y, z).$$

Here a three-place predicate is used to represent a function with two inputs, *i.e.:*

$$Add(x, y, z) \text{ true} \iff x + y = z.$$

So any axioms we want to use need to be translated from the usual operator notation. Unfortunately this is more than a simple rewriting exercise, and often involves increasing the complexity of the formula. Considering axioms related to groups, some are straightforward to translate. For example, the closure axiom is essentially unchanged. For an operator $*$, the usual notation for closure

$$\forall x \forall y \exists z \; x * y = z$$

becomes (keeping the same symbol for the predicate)

$$\forall x \forall y \exists z \;\; *(x, y, z).$$

For commutativity, the usual representation is

$$\forall x \forall y \; x * y = y * x$$

but using predicate notation requires introducing a third variable:

$$\forall x \forall y \forall z \;\; *(x, y, z) \equiv *(y, x, z).$$

If this seems unduly strong, note that the correct quantifier for the new variable z is $\forall$, not $\exists$. While we only expect a single "correct" value of z for given x and y, commutativity requires that, for *all* values of z, both sides of the biconditional have the same truth value. For *any* value of z, either both $*(x, y, z)$ and $*(y, x, z)$ hold, or neither of them does. If one holds while the other does not, then $*$ cannot be commutative.

By far the most difficult case is associativity, which merits a brief digression. The usual presentation is:

$$\forall x \forall y \forall z \; x * (y * z) = (x * y) * z.$$

The predicate representation is not immediately clear; one obvious but incorrect approach is to try to start nesting terms inside each other, but this confuses predicates with function symbols. We need a representation with predicates only, which requires a more roundabout approach.

Ignore the quantifiers for the moment, and define new variables:

$$x * \underbrace{(y * z)}_{=u} = \underbrace{(x * y)}_{=v} * z$$

So we have the following:

$$\begin{aligned} y * z &= u \\ x * y &= v \\ x * u &= v * z \end{aligned}$$

Let $w = x * u = v * z$. That gives us these facts, in predicate notation:

$$\begin{aligned} &*(y, z, u) \\ &*(x, y, v) \\ &*(x, u, w) \\ &*(v, z, w) \end{aligned}$$

and associativity requires that if the first two hold, then the last two should either be both true or both false:

$$\forall x \forall y \forall z \forall u \forall v \forall w \ (*(y, z, u) \ \& \ *(x, y, v)) \supset (*(x, u, w) \equiv *(v, z, w)). \tag{4}$$

If again this seems unduly strong, recall the reasoning for the commutativity case above. The biconditional on the right needs to hold for *all* values of w, not merely the single "correct" answer for given x, y, z, u, v. Similarly u and v need to be universally quantified also.

Currently the program checks 15 common axioms, all involve a single relation only. All relations in the input are checked, the only restrictions are on their arity: some axioms require a relation of arity three (shown as arbitrary relation P below), others require arity two (shown as R).

Axioms from arithmetic:

commutativity: $\forall x \forall y \forall z \ P(x, y, z) \equiv P(y, x, z)$

associativity: (as above)

closure: $\forall x \forall y \exists z \ P(x, y, z)$

left identity: $\exists x \forall y \ P(x, y, y)$

right identity: $\exists x \forall y \ P(y, x, y)$

left identity, left inverse: $\exists x \forall y \exists z \ P(x, y, y) \ \& \ P(z, y, x)$

left identity, right inverse: $\exists x \forall y \exists z \ P(x, y, y) \ \& \ P(y, z, x)$

right identity, left inverse: $\exists x \forall y \exists z \ P(y, x, y) \ \& \ P(z, y, x)$

right identity, right inverse: $\exists x \forall y \exists z \ P(y, x, y) \ \& \ P(y, z, x)$

Axioms from geometry:

antisymmetry: $\forall x \forall y (R(x,y)\ \&\ R(y,x)) \supset (x = y)$

congruence: $\forall x \forall y\ R(x,y) \equiv R(y,x)$

transitivity of congruence:
$\forall x \forall y \forall z \forall u \forall v \forall w (R(x,y) \equiv R(z,u)\ \&\ R(x,y) \equiv R(v,w)) \supset (R(z,u) \equiv R(v,w))$

Axioms from equivalence relations:

reflexivity: $\forall x\ R(x,x)$

symmetry: $\forall x \forall y\ R(x,y) \equiv R(y,x)$

transitivity: $\forall x \forall y \forall z\ (R(x,y)\ \&\ R(y,z)) \supset R(x,z)$

These axioms are produced in both typed and untyped versions. Some of the typed versions make broad assumptions about the possible arrangement of types in order to avoid generating too many formulas. In congruence for example, both x and y wind up occupying both left and right sides of the predicate variable tuple (we have both $R(x,y)$ and $R(y,x)$) so they are assumed to be the same type.

7 Source file directory

```
src
|--aa
| |-- AaGUI.java                      GUI functions, main method, overall control flow
| |-- And.java                        tree node for &
| |-- CommonAxiom.java                decorator class, adds a text label to FormulaTree
| |-- Equals.java                     equals operator, appears only in prover files
| |-- ExistentialQuantifier.java      tree node for ∃ + variable
| |-- FormulaGenerator.java           Generates all possible formulas
| |-- FormulaTree.java                expression tree for first-order logic
| |-- HandElimination.java            static methods for redundancy elimination
| |-- Iff.java                        tree node for ≡
| |-- Implies.java                    tree node for ⊃
| |-- Input.java                      stores all data from input file
| |-- Node.java                       abstract parent class for tree nodes
| |-- Not.java                        tree node for ¬
| |-- Operator.java                   abstract parent class for logical operators
| |-- Or.java                         tree node for ∨
| |-- ParallelMace4Call.java          Java Runnable object for Mace4 process
| |-- ParallelProver.java             handles parallel Prover9 / Mace4 calls
| |-- ParallelProver9Call.java        Java Runnable object for Prover9 process
| |-- Predicate.java                  tree node for predicate terms
| |-- Prefix.java                     quantifier prefix
| |-- ProverSearchResults.java        enum for prover return values
| |-- Quantifier.java                 abstract parent class for quantifiers
| |-- Relation.java                   details of a particular relation in the input
| |-- Type.java                       types for typed input elements
| |-- TypedExistentialQuantifier.java existential quantifier with typed variable
| |-- TypedFormulaGenerator.java      typed formula generator
| |-- TypedQuantifier.java            abstract parent class for typed quantifiers
| |-- TypedUniquenessQuantifier.java  uniqueness quantifier with typed variable
| |-- TypedUniversalQuantifier.java   universal quantifier with typed variable
| |-- UniquenessQuantifier.java       tree node for ∃! + variable
| |-- UniversalQuantifier.java        tree node for ∀ + variable
| |--WildType.java                    an unconstrained type
| |--WitnessSet.java                  debugging class
```