Programming Research Group

POLYANNA TECHNICAL MANUAL
(VERSION 1.00)

Richard Gault

PRG-RR-01-20

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Polyanna is a program for finding the polymorphisms of a given set of finite relations over a finite domain. Other programs have been written in the past to solve this problem, but all have run prohibitively slowly. Polyanna is the first program to fully exploit the symmetry inherent in this problem, and as a result, can run exponentially faster than its rivals.

This manual details the working of Polyanna at several levels of abstraction. We begin by describing the program from a user's point of view, and then give a detailed account of Polyanna's internals, which will be of interest to those who wish to modify or extend the program. We conclude with a high-level discussion of some of the algorithms used in Polyanna.

1 Introduction

Polyanna is a program for finding the polymorphisms of a given set of relations over a finite domain. It differs from other such programs, notably Closure, in that it tries to exploit as much symmetry as it can when it performs the polymorphism search. The ideal of the program is to both perform the search, and display the results in such a way that all "similar" polymorphisms are considered as one. In general, this will have the effect of speeding up the program by an exponential factor. We thus strive to recognise and exploit symmetry wherever we can.

As a trivial example, if one were to feed the "allows anything" relation (for some domain) into the program, it would spot this and immediately return a specification of all possible polymorphisms over that domain. No search is required; the calculation would be essentially instantaneous whatever the size of the input domain.

Of course, the program will never be able to recognise every possible form of symmetry, and many inputs will still cause it to take an unreasonable amount of time. It would be excessively optimistic to imagine that a program could be written which would always be able to find all polymorphisms in a tractable manner. The program's name is a contraction of 'Polymorphism Analyser', but the pun on Pollyanna is deliberate.

This document gives an account of the internal workings of the program at several levels of abstraction. It also describes its user interface. Because Polyanna is a continuously evolving program, this documentation is forever in a state of flux. In addition, it may sometimes not accurately reflect the current internal workings of Polyanna. The author would be grateful to hear of any errors or omissions, as well as of any bugs in Polyanna itself. He may be contacted via the following web page, from which the latest editions of the program, and this manual, are always available.

`http://www.comlab.ox.ac.uk/oucl/research/areas/constraints/`

This manual has been written primarily to aid those who wish to modify or extend Polyanna; consequently, many portions of it are technical and abstruse, and of very limited interest to the general reader. The reader who just wishes to learn more about the high-level algorithms employed should probably read the accompanying paper [1] instead. This would also be a good introduction to the algorithms employed for those who intend to read the present document.

Polyanna is a command-line program written in C++. The document provides:

- The user interface to the program
- A description of the overall structure of the program
- A high-level overview of the function of each class

- A detailed description of the public interface to each class
- A list of the exceptions which each class might throw
- A detailed description of the private functions and variables in each class
- A discussion of any non-trivial algorithms used.

Throughout this document, any text which is placed within square brackets, and italicised *[*** thus ***]* indicates a marginal comment which may be omitted at first reading. Typically, such comments suggest possible future extensions which may be made to the program: it is convenient to have these extensions mentioned as they appear, rather than clustered in one place at the end of the manual.

2 The User Interface

In this section we describe how the program operates from a user's point of view. We assume that the reader is familiar with the concept of polymorphisms, and the application of universal algebra to constraint satisfaction. For more details, see [2].

2.1 Running the Program

We will assume that Polyanna is being run from the command-line on a terminal. At the time of writing, this is the only mode of execution. However, a windowed interface may become available in the future.

To execute the program, type

```
polyanna [<options>] <filename> [<filename> [<filename> ...]]
```

where each of the `<filename>`s refers to an input file containing one or more relations. The program will find the polymorphisms of the set of all relations in all supplied files. The domain of the polymorphisms is taken to be the union of the domains of the relations in this set: there is currently no way of specifying it to be a superset of this. The arity of polymorphisms searched for is given by a command-line option. The meaning of all the options, and the format of the input files is given below.

The output of the program will normally be a list of *function specifications*, each of which defines one or more polymorphisms of the input relations. As a simple example, if we search for all idempotent polymorphisms of the simple relation $R = \{\langle 0\rangle, \langle 1\rangle\}$ (over the domain $\{0, 1\}$), then the function specification which is output is:

$$\begin{array}{rcl} f(0,0) & = & \{0\} \\ f(0,1) & = & \{0,1\} \\ f(1,0) & = & \{0,1\} \\ f(1,1) & = & \{1\}. \end{array}$$

This specification induces four functions in the natural way: each of $f(0,1)$ and $f(1,0)$ can take either the value 0 or the value 1 independently of each other. These four functions are all idempotent polymorphisms of R. By default, the program guarantees that each of the functions induced by a specification will be a polymorphism of the input relations which also satisfies any additional constraints imposed by the user. This behaviour can be changed, if desired, using the `--force` option described below.

Note that there is no way to directly specify the arity of an input relation to Polyanna. In all non-trivial cases it is obvious. The only time ambiguity arises is when Polyanna is supplied the empty relation {} as input. Polyanna treats the empty relation as being closed under every function (that is, essentially as being of arity 0).

A function specification which defines exactly one function (that is, in which every set has size one) is termed *complete*.

2.2 The Command-Line Options

There are quite a lot of command-line options. Most are available in both a short form (one letter, beginning with a `-`), and a long form (typically a word, beginning with `--`). A few options are only available in long form.

Many of the options are Boolean flags. A flag, such as the option to ensure that all functions found are idempotent, can be be forced on by using a notation such as `-I`, and turned off by using `-I0` (or `-~I`, if you prefer).

The full list of options currently implemented is as follows.

- `--arity` $\langle n \rangle$ or `-a` $\langle n \rangle$

  Search for polymorphisms of arity `n`. The default value depends on the other options supplied on the command-line. If you specify that all polymorphisms must be, say, associative then the arity will be automatically set to two, since associativity is only defined for binary functions. The same goes for all command-line options below which have a specific arity associated with them.

  If no arity is specified, either explicitly or implicitly, then its value is taken to be two.

- `--permopts` or `-p`

  List each polymorphism only once up to permutation (rather than displaying essentially the same function numerous times). It is a symmetry-breaking constraint. Turning this option off is likely both to increase the search time, and increase the number of solutions found. It is switched on by default under most circumstances, but turned off if searching for polymorphisms which are affine functions, Mal'tsev functions, or commutative functions. The reason for turning it off in the former two cases is that affine and Mal'tsev functions are not closed under permutations: so restricting our search to just a single permutation of each function is likely to cause us to miss several functions. The reason for turning the option off when searching for commutative functions is that commutative functions are precisely those arity-2 functions which are unchanged under permutation. Consequently, the setting of the option would have no effect, and we therefore turn it off since this makes the search run very slightly faster.

- `--automorphisms` or `-m`

  Factors out symmetries due to automorphisms (that is, arity-1 polymorphisms). The default is on. This option only factors out the application of an automorphism to both sides of a defining function simultaneously. To factor out the application of an automorphism to the left and/or right sides only, it is sufficient to additionally restrict the search to idempotent functions, using the `-I` switch (below). This is a stronger condition than factoring out automorphisms from the left and/or right, it is faster to execute, and it has been proved that when searching for polymorphisms which guarantee tractability, it is sufficient to restrict the search to idempotent polymorphisms.

*[*** A future version of Polyanna may allow the user to apply automorphisms to the left and/or right side of a definition only. The code is present in the program, but currently disabled (except for internal use) since it clashes with certain other constraints, including the idempotent constraint. Safeguards along the lines of the* `permutation_closed()` *functions will have to be added to the program if users are to have access to the feature. ***]*

- `--essentiallynary` or `-e`

  Every polymorphism found must make essential use of each one of its arguments. This is another symmetry-breaking constraint, and it is always on by default.

- `--associative` or `-A`

  Search for associative functions only. (Only applies to functions of arity 2.) Default is off.

- `--commutative` or `-C`

  Search for commutative functions only. (Only has meaning when the arity is 2.) Default is off.

- `--idempotent` or `-I`

  Search for idempotent functions only.

- `--semilattice` or `--ACI`

  Search for functions which are simultaneously associative, commutative, and idempotent. Equivalent to `-A -C -I`. This only has meaning when we are searching for functions of arity 2.

- `--affine` or `-F`

  Search for affine functions. This only has meaning when we are searching for functions of arity 3. Specifically, the search is restricted to polymorphisms which satisfy the following three identities.

$$\begin{aligned} f(f(x,y,z),z,u) &= f(x,y,u) \\ f(x,y,z) &= f(z,y,x) \\ f(x,x,y) &= y \end{aligned}$$

- `--majority` or `--nearunanimity` or `--nuf` or `-M`

  Search for majority/near-unanimity functions only. The arity of functions searched for must be at least three.

- `--conservative` or `-T`

  All functions found must be conservative.

- `--maltsev` or `--mal'tsev` or `-V`

  All functions found must be Mal'tsev. This option only applies when we are searching for arity-3 functions. A Mal'tsev function is one which satisfies the identity

$$f(x,y,y) = f(y,y,x) = x.$$

- `--twosemilattice` or `-S`

  This option only applies when we are searching for arity-2 functions. It restricts the search to 2-semilattice functions: that is, functions which are commutative, idempotent, and which also satisfy the identity
$$f(x, f(y, x)) = f(y, x).$$

- `--reduce` or `-r` (or `--squash`)

  If this option is set, then before the search proper begins, we conduct a search for non-injective arity-1 polymorphisms ("domain-squashing functions"). If we find any, then they are used to squash the domain of the input relations as much as possible. This does not affect the tractability of the relations, but may speed up the search for polymorphisms.

- `--force` $\langle n \rangle$ or `-f` $\langle n \rangle$

  How hard should we try searching for polymorphisms? The value of `n` must be an integer from 0 to 3. The meaning of each value is as follows.

  0 Every function specification which is output must contain at least one polymorphism which satisfies all the constraints demanded by the command line.

  1 Every function induced by one of the function specifications which is output, is a polymorphism which satisfies all the constraints demanded on the command line – with the exception of constraints whose sole purpose is to break symmetry. Although some attempt is made to satisfy these constraints, no guarantee is made that every (or indeed any) function induced by a given function specification will satisfy the symmetry-breaking constraints.

  2 Every function which is induced by one of the function specifications which is output, is a polymorphism which satisfies all the constraints demanded on the command line – including the symmetry-breaking constraints.

  3 Every function specification output will be complete: that is, will define a unique mathematical function (which satisfies every constraint required). This is normally only useful when the user wishes to count how many functions satisfy some set of constraints. It can be many orders of magnitude slower than the other options.

  The default value for `--force` is 1.

- `--stoponfirst` or `-s`

  Stop the computation as soon as the first function specification is output. This is useful when just checking for the existence of a function which satisfies certain constraints.

- `--quiet` or `-q`

  Don't display the functions as they are found. Default is off.

- `--count` or `-c`

  Display a final tally of the number of function specifications output. Default is on.

- `--tractable` or `-t`

  Determine whether or not the input relations are tractable, and if so, whether they are PTIME, LOGSPACE, etc. Most other command line options are ignored if you select this. The default is off.

Once Polyanna has displayed its analysis of the tractability, it exits. The exit code which is returned to the operating system depends on whether or not the relations are tractable, according to the following tabel.

0 Polyanna cannot determine the relations' tractabililty.

1 The relations are solvable in LOGSPACE.

2 The relations are solvable in NL.

3 The relations are solvable in PTIME.

4 The relations are conjectured to be solvable in PTIME.

5 The relations are NP-complete.

An exit code of -1 indicates "None of the above", and should never occur.

An exit code of 10 indicates an error. if the input is well-formed, this should never occur.

- `--help` or `-h`

  Displays some help text.

- `--viewconstraints` or `--vc`

  Display (in plain English) all the constraints which are in place, before the search for polymorphisms begins. This is not always quite as trivial as it may seem, since many of the relations will have been decomposed before the search begins, leading to unexpected sets of constraints. The default is off.

  There is no short-form for this option.

- `--viewrelations` or `--vr`

  Display all the input relations in their fully decomposed form. The format is fairly self-explanatory; for details see Section 3.1.1. The default is off.

  There is no short-form for this option.

- `--solve`

  If unset, the search for polymorphisms will not be carried out. This is only useful if used in conjunction with the `--viewconstraints` or `--viewrelations` options. The default if on.

  There is no short form for this option.

- `--iterations` $\langle n \rangle$

  This option is only useful when timing the program on certain inputs. It carries out the main polymorphism search a total of `n` times. If this option takes a value greater than 1, the `--quiet` option is turned on, and the `--count` option is turned off. The default value is, of course, 1.

  There is no short form for this option.

- `--enumeraterelations` $\langle n \rangle$ or `--er` $\langle n \rangle$

  This option turns Polyanna from a polymorphism-finder into a relation-generator. If this option is given, Polyanna will enumerate all relations on a domain of size n, but listing each relation only once up to renaming of its domain arguments and permutation of its tuples. In addition, it will omit all relations containing one or more constant tuples. The reason for this slightly eclectic mix of conditions is that Polyanna then generates all the essentially different, non-trivial relations on a domain of size n.

Almost all other options are ignored when Polyanna is given this command line option. The only exceptions are the `-a` switch, which determines the arity of relation to search for, and the `-q` switch, which suppresses output of the relations (and is useful if you want to count the number of essentially different relations which will be generated). Note that no input file is needed if this switch is selected.

There is no short form for this option.

2.3 The Format of the Input File

The input files have a very simple format. A typical file might look like:

```
Triangle and square

{<1,2>,
 <2,3>,
 <3,1>}

{<1,2>, <2,3>, <3,4>, <4,1>}
```

The file opens with a title. This is optional: if the title is omitted, then a sensible default will be supplied automatically. The title may spread over several lines. Its end is signified by a relation beginning on a new line.

Relations are entered as shown above. They are enclosed in {set brackets}, and individual tuples are enclosed by <inequality signs>. Commas between tuples are optional (indeed, an extra comma before the first tuple, or after the last tuple is also tolerated). Commas within tuples are not optional. Whitespace is ignored throughout the file, so it may be laid out however you wish. Two common conventions are shown above.

3 The Low-Level Implementation

The rest of this document is given over to describing how Polyanna works. We begin by detailing all of the classes present in the program, giving their public interface, and also describing their private implentation. Throughout, we use `nat` as a synonym for `unsigned int`. This `typedef` is defined in `misc_headers.h`, and is the only systemic "language-change" to be employed in the program.

3.1 The `Relation` Class

3.1.1 Overview of the `Relation` Class

The `Relation` class is the probably the most important class in the program. It implements mathematical relations – that is, sets of tuples of domain elements. In fact, the implementation forces domain elements to be integers, but this is a minor restriction.

Every `Relation` has two domains associated with it: a real domain, and an apparent domain. The apparent domain is just the set of all domain elements which occur within the relation. The real domain (which is always a superset of the apparent domain) is the actual domain over which the relation is defined. As an example, consider the relation $R = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 2,2\rangle\}$. Its apparent domain is the set $\{1, 2\}$. If that is also its real domain, then R may be disregarded when it comes to finding polymorphisms – R is closed under all functions over $\{1, 2\}$. However, if the real domain of R is actually $\{1, 2, 3\}$ (say), then R cannot be completely disregarded. The best we can do is to replace R by the relation $\{\langle 1\rangle, \langle 2\rangle\}$.

Each `Relation` has a type, indicating how it may be decomposed, and a (possibly empty) set of child `Relations`. If a decomposition is possible, the `Relation`'s children are the simplified `Relation`(s) from which it is composed. The possible types are:

- `Unknown`. We have not yet analysed this relation to find its type.
- `TrivialIrreducible`. The relation has an arity or size of 0.
- `OtherIrreducible`. The relation is non-trivial, but we do not know of any way of simplifying it.
- `DuplicatedColumns`. The relation has two or more identical columns. It has exactly one child: the relation with the duplicated columns removed.
- `CartesianPower`. A special case of repeated `CartesianProducts`, but one which is easier to test for. A relation is a `CartesianPower` if it is of the form R^a, where R is an arity-1 relation, and a is an integer greater than 1. Relations which, in traditional parlance, are 'allows-anything' relations are `CartesianPowers` of arity-1 relations. The sole child of a `CartesianPower` relation is its associated arity-1 relation.
- `CartesianProduct`. The relation is the Cartesian product of two or more other relations. Its children are these other relations.
- `DisjointDomains`. The relation is the union of two or more relations which are over disjoint domains. Its children are these relations. (Note that at present only relations of arity at least 2 are considered to be unions of relations over disjoint domains. Arity-1 relations are excluded from the check, since they are always trivially disjoint unions of size-1, arity-1 relations. This behaviour is in contrast to the behaviour of the algorithm for checking whether a relation is a Cartesian product: that algorithm always splits up arity-1 relations where possible.)
- `AllowsAnything`. In normal terminology, an allows-anything relation can have any arity. Within the program however, terminology is slightly different. An `AllowsAnything` relation is, by definition, an arity-1 relation whose apparent domain is equal to its real domain. As mentioned above, all allows-anything relations (in the traditional sense) of arity greater than 1 are `CartesianPowers` of an `AllowsAnything` relation (in the present sense). An `AllowsAnything` relation has exactly one child: an empty relation (of type `TrivialIrreducible`).

Apart from its type, each relation can have several other attributes:

- A relation is `redundant` if its polymorphisms are the same as the polymorphisms of its set of children. For example, a `CartesianProduct` relation is `redundant`; a `DisjointDomains` relation is not.
- A relation is `top_level` if it is not a child of any other relation.

3.1.2 Public Interface to the `Relation` Class

We now describe, in some detail, the public interface to the `Relation` class.

- `Relation(nat a = 2);`

  The constructor. The arity of the relation may be supplied if desired, but this is not normally necessary since whenever a tuple is added to an empty relation, the arity of the relation is set to be the arity of that tuple.

- `void change_arity(nat a);`

  Change the relation's arity. If the relation is non-empty, and the new arity is different to the old one, an `AritiesInconsistent` exception is thrown.

- `nat get_arity() const;`

  Get the relation's arity.

- `nat get_size() const;`

  Get the relation's size (*i.e.*, the number of tuples it contains).

- `std::set<int> get_apparent_domain() const;`

  Get the apparent domain of the relation. This is recomputed every time the function is called, so is relatively slow. It could be cached if this were thought to be necessary.

- `nat get_apparent_domain_size() const;`

  Get the size of the apparent domain. This calls `get_apparent_domain()`, and so is slow.

- `void set_real_domain(const std::set<int> &s);`

  Set the real domain of the relation. This is automatically propagated to any children of the relation. If the domain supplied is smaller than the apparent domain, a `DomainError` exception will be thrown.

- `std::set<int> get_real_domain() const;`

  Get the real domain once more. This is fast, since the real domain is stored internally.

- `nat get_real_domain_size() const;`

  Find the size of the real domain. Again, this is fast.

- `bool add_tuple(const std::vector <int> &t);`

  Add the supplied tuple to the relation. If the tuple has an arity different to the arity of the relation (and the relation has size > 0) then an `AritiesInconsistent` exception is thrown. The function returns `true` if the tuple is new, and `false` otherwise.

  It is illegal to add a tuple to a relation of type other than `Unknown`. A `BelatedAlteration` exception is thown in such a circumstance. (This restriction, and the others to do with the `BelatedAlteration` exception could be circumvented if really necessary, by making the relation throw away its children and decompose itself once more. But this is a fair amount of work (especially since the children might be entries in a `RelationStore` in their own right), and I doubt that it would ever be necessary in any case.)

- `bool remove_tuple(const std::vector <int> &t);`

  Remove the supplied tuple from the relation. Returns `true` if successful. Throws `AritiesInconsistent` if `t` is of the wrong arity.

  It is illegal to remove a tuple from a relation of type other than `Unknown`. A `BelatedAlteration` exception is thown in such a circumstance.

- `Relation* make_top_level_copy();`

  Creates a new relation, whose tuples are the same as the tuples in the present relation. The only difference between the two relations is that the new one has a type of `Unknown`, is always a top-level relation, and so on. That is, it is a new, top-level relation is its own right, with no children, and which is undecomposed.

- `void construct_constraints(ConstraintStore& cs);`

  Generates a set of constraints which between them state that any function which satisifies every constraint in the set must be a polymorphism of the present relation. (The arity of the polymorphism is determined from `cs` itself; we shall see later that each `ConstraintStore` has a unique arity.)

- `bool decompose();`

  Relations aren't decomposed automatically. This decomposes a relation, setting its type, and constructing its children automatically. `decompose()` is then recursively called for any child relations formed. The function returns `false` if the relation is irreducible (that is, of type `TrivialIrreducible` or `OtherIrreducible`), and `true` otherwise.

- `void squash_with_map(const std::map<int,int>& squash_map);`

  The supplied `map` defines a unary function from the real domain of the relation to a strict subset of that domain. We apply the map to the each domain element in the relation, thus reducing our real domain. In addition, we call `squash_with_map()` for each of our children (if any).

- `std::set<Relation*> get_children() const;`

  Return our set of children.

- `nat number_of_children() const;`

  Return the number of children we have.

- `bool get_top_level();`

  Returns `true` if, and only if, the present relation is a `top_level` relation. (See above for the definition of a `top_level` relation.)

- `bool get_redundant() const;`

  Returns `true` if, and only if, the present relation is `redundant`. (That is, could safely be deleted: see above for the precise definition.)

- `enum RelationType {Unknown, TrivialIrreducible, DuplicatedColumns, CartesianPower, OtherIrreducible, CartesianProduct, DisjointDomains, AllowsAnything, ConjunctiveProduct};`

  Possible relation types.

- `RelationType get_type() const;`

  Get the type of the current relation.

- `void hierarchical_output(std::ostream& s) const;`

  Displays the relation as a fully-bracketed expression, indicating how it (and, recursively, its children) can be decomposed into irreducible relations.

- `void output_as_tuples(std::ostream& s, bool new_lines = true) const;`

  Displays the relation (but not its children) directly as a set of tuples. If `new_lines` is set to `true`, then the relation will be nicely formatted with one tuple per line. If not, then the relation will be formatted with just a comma and a space between tuples.

3.1.3 Exceptions of the `Relation` Class

We now come to the exceptions which may be thrown by member functions of the `Relation` class.

- `class AritiesInconsistent{};`

  There has been a mix-up over the arities of the relation.

- `class SanityError{};`

  An internal sanity check has been violated. This should never be thrown.

- `class DecompositionError{};`

  A sanity check to do with decomposition functions has been violated. This should never be thrown.

- `class ReadError{};`

  An error occurred when reading in a relation.

- `class DomainError{};`

  The real domain is smaller than the apparent domain.

- `class BelatedAlteration{};`

  At attempt has been made to alter a relation which has already been analysed and decomposed.

3.1.4 Friends of the `Relation` class

Not member functions as such, but it's worth mentioning the following two functions.

- `friend std::ostream& operator<< (std::ostream& s, const Relation& r);`

- `friend std::istream& operator>> (std::istream& s, Relation& r);`

  These provide input and output of relations. The output is presented as a list of tuples (it just calls `output_as_tuples(s, true)`). The input format for relations is fairly generous in what it will allow. The arity and size of the relation are determined automatically. Whitespace is ignored. Commas between tuples are optional. (Indeed, one may also put a comma immediately after the opening { bracket, or immediately before the closing } bracket). A typical input relation might look like $\{\langle 0,1\rangle,\langle 2,3\rangle,\langle 3,4\rangle,\langle 4,1\rangle\}$. Note that a relation which is output via `output_as_tuples` is already in the correct format to be read in again. Whilst reading in a tuple, an `AritiesInconsistent` or a `ReadError` exception may be thrown.

3.1.5 Private Members of the `Relation` Class

So much for the public interface to the Relation class. What about its private variables and member functions?

- `nat arity;`

  The arity of the relation, of course.

- `std::set < std::vector<int> > set_of_tuples;`

  The tuples within the relation.

- `typedef std::set< std::vector<int> >::const_iterator RI;`

  Just a typedef for the iterator for `set_of_tuples`.

- `std::set<int> real_domain;`

  The real domain.

- `std::set<Relation*> children;`

  Pointers to our children (if any).

- `Relation* make_child(Relation* copy_from);`

  Create a new child of the present relation (and insert it into our `children` set). The child takes its `set_of_tuples` from the relation `copy_from`. Its real domain is taken from the present relation. If this is not a superset of the apparent domain of `copy_from` then a `DomainError` exception is thrown. If `copy_from` is `NULL`, then the child begins with an empty `set_of_tuples`.

- `bool redundant;`

- `bool top_level;`

- `RelationType type;`

- `bool squashed;`

  Variables holding information as to whether the relation is `redundant` or `top_level`, and also its type. If the relation has had a squashing function applied to it, the `squashed` will be set to `true`.

- `typedef bool (Relation::* DecompositionFunction) ();`

  The type of the various decomposition functions, below.

- `bool apply_decomposition(DecompositionFunction decomposer);`

  This applies the specified decomposition function in order to try to decompose the relation. Returns `true` if a decomposition is performed, or `false` if the relation is `OtherIrreducible` or `TrivialIrreducible`. A `DecompositionError` exception may be thrown if certain sanity checks fail.

  If a successful decomposition takes place, then any children are automatically decomposed also.

- `bool check_for_trivial_relations();`

- `bool check_for_duplicated_columns();`
- `bool check_for_allows_anything();`
- `bool check_for_cartesian_power();`
- `bool check_for_cartesian_product();`
- `bool check_for_disjoint_domains();`
- `bool check_for_conjunctive_product();`

  Various decomposition functions. Each of them tests the current relation to see whether it can be decomposed in the specified way. If so, they set the type of the relation appropriately, make it `redundant` if necessary, and create any required children (with the aid of `make_child()`, described above). They return `true` if they performed a succesful decomposition, and `false` otherwise. Note that `check_for_trivial_relations()` returns `true` if sucessful, yet `decompose()` returns `false` in this case. This is because `decompose()` does not consider spotting that a relation is trivial to be a decomposition.

  *[*** `check_for_conjunctive_product()` is not yet implemented. ***]*

  A couple of member functions are just helper functions for one or more of these decomposition functions:

- `nat delete_columns(const std::vector<bool>& keep);`

  `keep` is a vector of the same arity as the present relation (else `AritiesInconsistent` is thrown). All columns (from `set_of_tuples`) of the relation for which `keep[col] == false`, are deleted. The number of columns deleted is returned.

- `bool check_split_for_cartesian_product(const std::vector<bool>& split);`

  `split` is a vector of the same arity as the present relation (else `AritiesInconsistent` is thrown). If we split the relation into two, by putting columns for which `split[col]` is `true` into one relation, and columns for which `split[col]` is `false` into the other relation, does the Cartesian product of the two resulting relations equal the present relation? If so, make the two relations defined by `split` into children of the present relation, set the type of the current relation to be `CartesianProduct`, and make the present relation `redundant`. Return `true` if (and only if) we do this.

3.2 The `RelationStore` Class

3.2.1 Overview of the `RelationStore` Class

A `RelationStore` is, at its simplest, just a set of relations (or, more accurately, a set of pointers to `Relation` objects). To make it more useful however, it is augmented with several member functions which allow the relations within the store to be manipulated in a uniform way. It also has the capability to read in relations from a file, and add them to the store.

Because a `RelationStore` is a set of pointers to relations, a relation may occur more than once in any store (at a different address for each occurrence.)

Each `RelationStore` has a title. It may be set and read by the user. If none is set, then it has the default fall-back “`Untitled set of relations`”. When a set of relations is read in from a file, any lines of text which occur before the first relation are taken to be a title for the relations in that file. If no title is given in the file, then the file is taken to have the title “`Untitled relations` from file <filename>". The title of the RelationStore is taken to be the title given in the file (if the store was previously untitled), or "<previous title>\n and <title given in file>" (if the store already had a title).

The RetaltionStore class is a subclass of the SolutionReceiver class (see below). This allows member functions such as squashing_polymorphism_receiver() to receive infomation from the solver about arity-1 homomorphisms.

3.2.2 Public Interface to the RelationStore Class

The public member functions of RelationStore are as follows.

- `RelationStore();`

  Just a constructor. It takes no arguments.

- `void add_relation(Relation* rp);`

  Put a new relation into the store.

- `bool is_pointer_in_store(Relation* rp) const;`

  Check whether the supplied relation is already in the store. Note that because the store is a set of pointers to relations, this function actually tells us whether the supplied pointer is already in the store.

- `nat get_size() const;`

  Return the number of distinct pointers in the store.

- `std::set <int> get_apparent_domain() const;`

  This calculates the apparent domian of the RelationStore: that is, the union of the apparent domains of all the individual relations in the store. Because the apparent domains of all the individual relations in the store are calculated on-demand, this can be a very slow operation (relatively speaking). It should be used sparingly.

- `nat get_apparent_domain_size() const;`

  This returns the size of the apparent domain of the RelationStore. It makes use of get_apparent_domain, and hence is very slow.

- `void set_real_domain(const std::set<int> &s);`

  Set the real domain of the store to be the set s. If s is not at least as large as the apparent domain, a Relation::DomainError exception will be thrown. The real domain is propagated to all relations currently in the store. (But if a relation is subsequently added to the store, it doesn't automatically get given the real domain of the store.)

- `std::set<int> get_real_domain() const;`

  Get the real domain of the store. Note that this is fast, since the real domain is stored in a variable.

- `nat get_real_domain_size() const;`

  Find the size of the real domain. Fast.

- `void read_file(const std::string filename);`

  Read the file given by `filename` into the store. The title of the store will be modified appropriately, as discussed above. If something goes wrong when reading the file, an `ErrorOpeningFile` or an `ErrorReadingFile` exception will be thrown. These exceptions both indicate the filename of the problematic file. The format of the input file has already been implicitly defined in these notes (A title, followed by one or more relations, in the format discussed in the description of the `Relation` class.) It is explicitly described in Section 2.3 above.

- `void set_title(const std::string t);`

  Set the tile of the store, over-riding any titles already computed. Note that the title may be modified, as described above, if the contents of a file is later added to the store.

- `std::string get_title() const;`

  Read the title. Returns "`Untitled set of relations`" if no title has yet been defined.

- `Relation* extract_relation();`

  Chooses an arbitrary relation from the store, removes it, and returns a pointer to that relation. If the store is empty, then `NULL` is returned.

- `bool decompose_all_relations();`

  Decompose all relations in the store. (That is, call `decompose()` for each relation in the store.)

- `bool squash_relations(bool quiet);`

  Find a a squashing function, and apply it to all the relations in the store. The squashing function applied is chosen according to two criteria. First, it must squash the domain as much as possible. Second, amongst all maximally-squashing functions, the one applied must maximise the number of domain elements which are mapped to themselves. This function works hand-in-glove with `squashing_polymorphism_receiver()`.

- `nat get_injective_unary_polymorphisms(`
  `std::set<std::map<int, int> >* mappings) const;`

  Find all injective unary polymorphisms of the relations in the store, and place them into the `map` supplied. The number of polymorphisms found is returned as a `nat`. This function works closely with `injective_polymorphism_receiver()`.

- `void add_polymorphism_constraints(ConstraintStore& cs) const;`

  For each non-redundant relation, R, in the store, this generates several constraints which between them state that any functions which satisfy all of the constraints must be a polymorphism of R. These constraints are then added to the `ConstraintStore cs`.

  In essence, it just calls `R.construct_constraints()` for every relation in the store.

- `enum Tractability {Logspace, NLogspace, PTime, NPComplete, ConjecturedPTime, Unknown};`

  Different levels of tractability of the constraint satisfaction problem. `Logspace` is used for constant-closure; `NLogspace` is currently unused; `PTime` is used for closure under semilattice operations, affine operations, and majority operations; `ConjecturedPTime` is used for Mal'tsev closure; `NPComplete` is used for relations which when reduced are only closed under essentially unary operations; `Unknown` is used for all other relations.

- `Tractability RelationStore::test_tractability(bool quiet = false);`

  Try to determine whether the relations in the store can be proved to be tractable. It checks to see whether the relations are all closed under a constant operation, a semilattice operation, a Mal'tsev operation, or a near-unanimity operation. If there is essentially only one relation in the store, and if it is a disjoint-domains relation or a conjunctive product relation, then the component realtions will be tested for tractability individually.

  The function returns the tractability level of the relations – see `Tractability` above. If the variable `quiet` is set to `false`, then the user will be kept informed as the progress of the test.

- `void hierarchical_output(std::ostream& s) const;`

  Display, on stream `s`, all the `top_level` relations in the store as fully-bracketed hierarchical expressions. (That is, call `hierarchical_output(s)` for each `top_level` relation in the store.)

- `void output_as_tuples(std::ostream& s) const;`

  Display, on stream `s`, all relations (`top_level` or otherwise) which are in the store. The relations are all displayed as sets of tuples.

3.2.3 Exceptions of the `RelationStore` Class

We now come to the various exceptions which are defined by the `RelationStore` class.

- `class SanityError{};`

  This should never occur.

- ```
  class ErrorOpeningFile
  {
  public:
  std::string filename;
  ErrorOpeningFile(std::string fn) {filename = fn;}
  };
  ```

  File not found. `filename` contains the filename we we looking for.

- ```
  class ErrorReadingFile
  {
  public:
  std::string filename;
  ErrorReadingFile(std::string fn) {filename = fn;}
  };
  ```

  The input file given by `filename` contains a bug.

3.2.4 Friends of the `RelationStore` Class

There is only one friend: the output function.

- `friend std::ostream& operator<< (std::ostream& s, const RelationStore& rs);`

  This function simply outputs the contents of the store in a hierarchical fashion (*i.e.*, using `hierarchical_output()`).

3.2.5 Private Members of the `RelationStore` Class

Finally, we describe the private member functions and variables of the `RelationStore` class.

- `std::set <Relation*> store;`

  The set of pointers to relations which makes up the store itself. The heart of the class.

- `typedef std::set<Relation*>::const_iterator RSI;`

  Just a typedef for the iterator over members of the store.

- `std::string title;`

  The title of the store. As indicated above, this is normally set automatically.

- `bool title_given;`

  `false` initially (when the title is just "`Untitled set of relations`"). `true` as soon as a title has been implicitly or explicitly given.

- `static const nat max_line_length = 256;`

  This is just a kludge to save problems when reading in a store title from a file. No title line in the input file can be more than 256 characters long. (Note that an actual title can be longer than this – it's just that it must be put onto two or more lines.) The reasons for this are explained in a comment in the implemetation of `RelationStore::read_file()`.

- `std::set<int> real_domain;`

  The real domain of the `RelationStore`. Should always be a superset of the apparent domain.

- `std::map<int, int> squash_map;`

- `nat squash_range_size;`

- `nat squash_unchanged_size;`

  If we're searching for squashing functions, then these three variables are used to keep track of the best squashing function found so far. `squash_map` is the squashing function itself, `squash_range_size` is the size of the function's range (this must be minimised), and `squash_unchanged_size` contains the number of domain elements which are mapped to themselves. This latter must be maximised, subject to `squash_range_size` being minimised.

- `void squashing_polymorphism_receiver(const Function& f);`

  If we're searching for squashing functions, then this function is called every time the solving routine finds a new non-injective unary polymorphism. It updates the above three variables accordingly.

- `mutable std::set<std::map<int,int> >* iup_maps;`

- `void injective_polymorphism_receiver(const Function& f) const;`

  If we're searching for injective unary polymorphisms, this function is called every time a new polymorphism is found. It adds each new polymorphism to the map `iup_maps`.

- `bool flatten_store();`

- void flatten_store_helper(std::set<Relation*> *rel_setp, Relation* rp);

  For each relation in the store, traverse its decomposition tree, and add to the store any relations which aren't already present. (Because the store is a *set* of pointers, if this function is called several times in succession, only the first call will have any effect.)

  flatten_store() is the function which should be called to do the work; the function flatten_store_helper() is just a helper function.

- bool test_for_constant(bool quiet = false)
- bool test_for_aci(bool quiet = false);
- bool test_for_bcc(bool quiet = false);
- bool test_for_affine(bool quiet = false);
- bool test_for_maltsev(bool quiet = false);
- bool test_for_nuf(bool quiet = false);
- bool test_for_essentially_unary_only(bool quiet = false);

  Test to see whether the relations in the store are closed under any constant functions (resp. ACI functions, binary commutative conservative functions, etc.) Each function returns true if it is closed under such a function, and false otherwise. The function bool test_for_essentially_unary_only() returns true if, and only if, the relations in the store are not closed under any functions of essential arity 2 or more.

3.3 The FunctionArguments Class

3.3.1 Overview of the FunctionArguments Class

A traditionally defined function is a map from tuples of domain elements, to domain elements. Consequently, one might expect Polyanna to contain a function class, whose principal component was a map of type map<vector<int>, int>. Function assignments would then be written simply as f[t] := v, where t is some tuple.

The problem with this approach however, is that such assignments are inherently slow – they take $O(|\mathtt{t}|)$ steps to execute. In addition, we often want to ask questions of tuples. We might ask whether t is a constant tuple for example, or wonder what its i'th permutation is. Calculating the answers to these questions every time they're needed is prohibitively slow.

The FunctionArguments class is designed to circumvent these problems. Each different tuple has a unique FunctionArguments object associated with it. Because of this unique association, we can use pointers to FunctionArguments objects in place of tuples. Looking up the value of $f(\textit{some\_tuple})$ reduces to looking up the value of $f(\textit{some\_pointer})$ – a much faster operation. Similarly, testing for equality of tuples reduces to testing for equality of pointers. In addition, each FunctionArguments object has various snippets of information attached to it: such as the abovementioned "Is this tuple constant?" and "What are this tuple's permutations?". The answers to these questions then only have to be computed once.

Perhaps the most important function in the FunctionArguments class is the static function tuple_to_function_arguments(), which converts a vector of integers into a FucntionArgument object. If the FunctionArgument object corresponding to the supplied vector already exists, then its address is returned. Otherwise, a new FunctionArguments object is created, and the address of this new object is returned.

The `FunctionArguments` class has a second rôle also. We have already mentioned that looking up $f(some\_pointer)$ is a relatively fast operation if f is defined as a `map` from pointers to domain elements. Even faster than defining f as a `map` however, is for each `FunctionArguments` object to contain $f(that\_object)$ directly. The naïve implementation of this only allows us to have one function in the program. We can do better than this however, and associate with each `FunctionArguments` object a `vector` of results – one for each of several possible functions. Each function has a unique ID number: a positive integer wich is used to look up the value of f in that array. The functions `get_new_function_id()` and `destroy_function()` are called whenever a new `Function` object is created and destroyed, respectively. The former assigns a unique ID number to the function, the latter returns the ID number to a central pool for use by another new function. Needless to say, the pool of possible ID numbers automatically grows as necessary.

To make all of this work effectively, the `Function` class is a friend of the `FunctionArguments` class.

3.3.2 Public Interface to the `FunctionArguments` Class

The public interface to the class is as follows.

- `static const FunctionArguments* tuple_to_function_arguments(`
  `const std::vector<int> &t);`

  We have already discussed this function briefly. It translates the supplied tuple `t` into a `FunctionArguments` object, constructing the object if necessary.

- `void display(std::ostream &s) const;`

  An output routine. It displays the associated tuple on stream `s`.

- `void display_perms() const;`

  This function is for debugging; it will likely be removed in future. It displays all the permutations of the associated tuple on `cout`.

- `nat get_arity() const;`

  Returns the arity of the tuple we represent.

- `int operator[] (nat n) const;`

  A way of accessing the elements of the tuple we represent, without having to copy the whole tuple back and forth.

- `std::vector<int> function_arguments_to_tuple() const;`

  If, on the other hand, you need access to the whole tuple, this function will give you a copy.

- `nat get_number_of_permutations() const;`

  Each `FunctionArguments` object stores a copy of all its permutations, in *some fixed order*. This is important, since the constraint which ensures that each function is only listed once up to permutation of its arguments, depends on this fact. So for some fixed arity, the first permutation might be "Swap the values stored in the first and second places in the tuple," for example. One consequence of enforcing this fixed order is that the number of permutations of a tuple depends only on the arity of the tuple. A tuple which has repeated permutations (an extreme example is a constant tuple, where all permutations are identical) will not have the repetitions removed.

Since the trival permutation is not counted, the value returned by this function should always be $a! - 1$.

- `const FunctionArguments* get_permutation(nat i) const;`

  Return the $(i+1)$'th non-trivial permutation. That is, treat the non-trivial permutations as if they are an array with an index beginning at 0.

- `const FunctionArguments* get_reverse_permutation() const;`

  Return a pointer to a `FunctionArguments` tuple which is the reverse permutation of the current tuple.

- `bool is_constant_function_arguments() const;`

  Returns `true` if, and only if, the associated tuple is a constant tuple: that is, of the form $\langle a, a, \ldots, a \rangle$. An empty tuple is treated as constant. The value of a can be determined using the `operator[]` function.

- `bool is_almost_constant_function_arguments(int& a, int& b, nat& bpos) const;`

  Returns `true` if, and only if, the associated tuple is almost constant: that is, of the form $\langle a, a, \ldots, a, b, a, \ldots, a \rangle$. Tuples of size two or less are *not* treated as almost constant. On the other hand, constant tuples (of arity at least three) are also almost-constant.

  The value contained in the variables `a`, `b`, and `bpos` are destroyed. If the tuple is almost constant, their new valuea are the almost-constant value (a, in the terminology of the previous paragraph), the remaining value (b), and the index of b in the tuple, respectively. In the case of a constant tuple, `a` will equal `b`, and `bpos` will be 0.

3.3.3 Exceptions of the `FunctionArguments` Class

The `FunctionArguments` class only has the one exception associated with it.

- `class SanityError{};`

  This is thrown, as one would expect, whenever we detect an internal error relating to the `FunctionArguments` class.

3.3.4 Friends of the `FunctionArguments` Class

Apart from two output functions, the whole of the `Function` class (to be discussed below) is a friend of the `FunctionArguments` class.

- `friend std::ostream& operator<< (std::ostream& s,`
  `const FunctionArguments& fa)`

  When passed a `FunctionArgument`, this function calls `FunctionArguments::display()` to output the associated tuple.

- `friend std::ostream& operator<< (std::ostream& s,`
  `const FunctionArguments* fa)`

  This function is similar, except that it is passed a pointer to a `FunctionArgument` object. The function automatically follows the pointer, and displays the tuple associated with the resulting `FunctionArguments`.

- `friend class Function {};`

  As mentioned above, this is needed to allow `FunctionArguments` objects to define functions, and enable speedy access to them.

3.3.5 Private Members of the `FunctionArguments` Class

We now come to the Private members.

- `std::vector<int> tuple;`

  The tuple we represent.

- `static std::map<std::vector<int>, const FunctionArguments*> tuple_to_fn_args_map;`

  This map is used internally to determine whether or not we've created a `FunctionArguments` object corresponding to any given tuple and, if so, what its address is.

- `void FunctionArguments_constructor(const std::vector<int> &t);`

  When a new `FunctionArguments` object is created, it must do some initialisation. Under normal circumstances, this would go in a constructor, but this causes problems in the current case. For one of the things a new `FunctionArgument` object does on creation is to store the addresses of the `FunctionArguments` corresponding to its permutations. This may well result in having to construct these other `FunctionArguments` first.

  Consider the consequences. We try to create object O_1. In the process, this tries to create object O_2. This in turn needs the address of object O_1. But since O_1 hasn't finished being created yet, we don't know its address, and so its address certainly can't have been added to `tuple_to_fn_args_map`. Consequenly, O_2 tries to initiate the construction of O_1. An infinite loop beckons.

  To avoid this, we don't let the `FunctionArgument` objects have constructors as such. Instead, whenever an object has been constructed, we immediately call this "pseudo-constructor". Since creation of `FunctionArgument` objects only occurs in one place (specifically, within `tuple_to_function_arguments()`) this does not cause significant problems.

- `std::vector<const FunctionArguments*> permutations;`

  A vector of permutations in some fixed order, as discussed above.

- `void generate_permutations();`

  Create the `permutations` vector.

- `const FunctionArguments* rev_perm;`

  The `FunctionArguments` whose associated tuple is the same as our own tuple, except backwards.

- `bool constant_function_arguments;`

  A variable which is `true` if, and only if, `tuple` is a constant tuple.

- `void initialise_constant_function_arguments();`

  Set up the above variable.

- `bool almost_constant_function_arguments;`

  A variable which is `true` if, and only if, `tuple` is almost-constant.

- `int acfa_major_value;`

  The almost-constant value (if `almost_constant_function_arguments` is `true`). 0 if it is `false`.

- `int acfa_minor_value;`

  The odd-value-out (if `almost_constant_function_arguments` is `true`). 0 if it is `false`.

- `nat acfa_minor_posn;`

  The index into `tuple` of the odd-value-out (if `almost_constant_function_arguments` is `true`). 0 if it is `false`.

- `void initialise_almost_constant_function_arguments();`

  Set up the above two variables.

We now come to those private member functions which implement the function look-up we described earlier. As mentioned, the `Function` class (to be defined below) is a friend of the present class, to enable it to access these functions. This has the unfortunate side-effect of allowing the `Function` class to access the whole of our internal representation. We could, perhaps, have made the `Function` class a subclass of the `FunctionArguments` class instead (thus allowing us to define the following few functions as `protected` members), but this would not have fitted the semantic meaning of a subclass.

- `static nat max_functions_definable;`

  A variable (initally 0, for convenience of initialisations) which holds the total number of functions which the `FunctionArguments` objects can define. It is automatically increased when necessary.

- `static std::set<nat> function_ids_available;`

  The set of functions IDs which are available to be handed out to newly created functions. The set of all IDs is always $\{0, 1, ..., $`max_functions_definable - 1`$\}$, so the current set is always a subset of this.

- `static nat get_new_function_id();`

  Takes an ID out of the pool, and returns it. If the pool is empty, then extend it. (Note: the initial values held by the `FunctionArguments` objects on behalf of the function with this ID are undefined (see `destroy_function()`), but the constructor of a `Function` object will always overwrite any spurious values already stored.)

- `static void destroy_function(nat id);`

  Return the supplied ID back to the pool. It doesn't destroy the values held by the function – we assume that a new function which is given that ID will store its own values.

- `void resize_functions_definable(nat new_size);`

  Forces an object to alter the number of functions it can define.

- `mutable std::vector<const FunctionValues*> apply_fn_result;`

  A `vector` whose *id*'th element is the result of applying the function whose ID is *id* to the present `FunctionArguments` tuple. We shall define the `FunctionValues` class presently; for now, it is enough to know that it holds the result of applying a function to a `FunctionArguments` tuple.

- `const FunctionValues* get_function_result(nat id) const`

  Reads the value of $f_{id}(fa)$, where *fa* is the current `FunctionArgument`.

- `void set_function_result(nat id, const FunctionValues* fv)`

  Sets the value of $f_{id}(fa)$ to be `fv`.

3.4 The `FunctionValues` Class

3.4.1 Overview of the `FunctionValues` Class

`FunctionValues` are to sets of integers, as `FunctionArguments` are to vectors of integers. That is, every set of integers has a unique `FunctionValues` object associated with it, whose purpose is partly to speed up the comparison and assignment of sets, and party to ensure that certain time-consuming operations are only done once each. The implementation is similar too. A set is converted into a pointer to a `FunctionValues` object using the `static set_to_function_values()` function, which returns the address of a previously created `FunctionValues` object if one exists, and creates a new such object if not.

There is a subclass of the `FunctionValues` class, called `EmptyFunctionValues`. This contains exactly one member object – the `FunctionValues` object corresponding to the empty set. The only reason it has been split off into a class of its own is that it improves efficiency slightly since we don't have to check for the empty set as a special case all the time in members of the regular `FunctionValues` class. The rest of this documentation will make no mention of this subclass, since it is invisible to the outside user.

3.4.2 Public Interface to the `FunctionValues` Class

- `static const FunctionValues* set_to_function_values(const std::set<int> &s);`

  The function which converts sets of integers into `FunctionValues` objects, constructing them if needed. We have discussed this function in some detail above.

- `std::set<int> function_values_to_set();`

  The converse function. Naturally it is rather slow, since the whole set has to be copied back to the calling function.

- `void display(std::ostream& s) const;`

  Output the set on output stream `s`

- `bool is_empty_set() const;`

  Is our set the empty set? (This function, like several others below, is in fact `virtual`, but we have already stated that this documentation will make no mention of the `EmptyFunctionValues` class.)

- `int min_value() const;`

  The minimum value in our associated set. Throws an `EmptySet()` exception if the set is empty.

- `int max_value() const;`

  The maximum value in our associated set. Throws an `EmptySet()` exception if the set is empty.

- `nat get_size() const;`

  The size of the associated set.

- `bool is_element(int e);`

  Is the integer `e` a member of the set we represent?

- `const FunctionValues* intersect(const FunctionValues* const other_fvp)`

  Find the intersection of our associated set, and the associated set of `other_fvp`. Ths intersection is, in fact, carried out by the static member function `find_intersection()` described below. Whenever an intersection is asked for, the result is stored away so that it can be quickly retrieved if the same intersection is asked for again later.

3.4.3 Exceptions of the `FunctionValues` Class

- `class EmptySet{};`

  Thrown when certain operations are carried out on an empty set.

- `class SanityError{};`

  In theory, never thrown. An internal error has produced an unexpected result.

3.4.4 Friends of the `FunctionValues` Class

As for the `FunctionArguments` class, we have two friends, both related to output.

- `friend std::ostream& operator<< (std::ostream& s, const FunctionValues& fv)`

  Display the `FunctionValues` set referred to by `fv` on stream `s`.

- `friend std::ostream& operator<< (std::ostream& s, const FunctionValues* fv)`

  Follow the `fv` pointer, and output the set associated with the resulting `FunctionValues` object.

3.4.5 Private Members of the `FunctionValues` Class

- `std::set<int> values;`

  The set we represent

- `static std::map<std::set<int>, const FunctionValues*> set_to_fn_values_map;`

  The map used to determine whether we've already created a `FunctionValues` object corresponding to the set of integers and, if so, what its address is.

- void FunctionValues_constructor(const std::set<int> &s);

  For similar reasons to the FunctionArguments class, it is unsafe to use real constructors when creating new FunctionValues objects. As before, since new objects are only created by the function const FunctionValues* set_to_function_values(), it does no harm to use this fake constructor instead.

- static const FunctionValues* find_intersection(const FunctionValues* fvp1, const FunctionValues* fvp2);

  Find the intersection (expressed as a pointer to a FunctionValues object) of the sets corresponding to the two supplied FunctionValues objects. The result of the intersection is saved away, so that the next time this intersection is asked for, we will be able to retrieve it very quickly.

- static std::map<std::pair<const FunctionValues*, const FunctionValues*>, const FunctionValues*> intersection_table;

  The table used to store intersections which we have already calculated. In the pair<const FunctionValues*, const FunctionValues*>, the first pointer is always less than or equal to the second pointer. This halves the size of table which we would otherwise need, and also ensures that having found $\S_1 \cap S_2$, we do not need to recalculate $S_2 \cap S_1$.

- int min_val;

- int max_val;

- nat size_of_set;

  Three variables which store the minimum value, the maximum value, and the size of the set values. (In the case of an empty set, the EmptyFunctionValues class is missing these three variables.)

3.5 The Function Class

3.5.1 Overview of the Function Class

There is little left to say about the Function class that has not already been said above. A function, intuitively, maps tuples of domain elements to sets of domain elements. Formally, it is little more than an index number supplied by the FunctionArguments class, which tells objects of that class where in an array they should expect to find the value of our function (applied to the set of arguments they represent). This value is in fact a pointer to a FunctionValues object, which represents a set of domain elements.

The Function class is a form of syntactic sugar which hides these ugly details from the rest of the program. The one useful thing which it does do is to provide the save_state() and restore_state() functions, which allow users to restore the function to a previous state it held. This is very useful when solving constraints, since it is very inefficient to pass a whole Function back and forth between successive levels of a backtracking recursion.

3.5.2 Public Interface to the Function Class

- Function(nat a, const std::set<int>& dom, const std::vector<const FunctionArguments*>& ord, bool partial = false);

The constructor. The arity, the domain, and a vector of all tuples to which this function applies should be supplied as arguments. The reason for using a vector rather than a set here, is that the vector gives the default order in which function tuples should be output when the function is displayed. Typically, this will be lexicographic order, but not necessarily. The `partial` argument merely indicates to the function whether or not it is a total function. In practice, the setting of this flag makes very little difference: it is merely the case that certain sanity checks which rely on the function being total will be omitted if it is known to be partial.

- `~Function();`

  Destructor.

- `Function* clone() const;`

  Create a new function, which is a copy of the present function (but without the saved history information).

- `void display_as_list(std::ostream &s,`
  `const std::vector<const FunctionArguments*> &ordering) const;`

- `void display_as_list(std::ostream &s) const;`

  Display the function – either using the default tuple ordering, or a custom ordering.

- `void display_as_relation(std::ostream &s,`
  `const std::vector<const FunctionArguments*> &ordering) const;`

- `void display_as_relation(std::ostream &s) const;`

  Display the function – either using the default tuple ordering, or a custom ordering – in the form of a relation. The tuple t is present in the relation iff $f(t) \neq 0$. This is mainly useful as part of the code for enumjerating essentially different relations. It probably has few uses outside this purpose.

- `const FunctionValues* get_value(const FunctionArguments* fa) const;`

  Returns the value of the function, when applied to the given `FunctionArgument`.

- `const FunctionValues* operator[] (const FunctionArguments* fa) const;`

  An abbreviation for the above. (Note that `[]` only provides read-access to the function.)

- `void set_value(const FunctionArguments* fa, const FunctionValues* fv);`

  Set the value of the function, when applied to `*fa` to be `*fv`.

- `void save_state();`

  Save away the current values of the function.

- `void restore_state();`

  Restore the function to the state it was in at the time of the last call to `save_state()`. Needless to say, calls to `save_state()` can be nested; successive calls to `restore_state()` will then restore the function to each saved state in turn. If more calls to `restore_state()` are made than have been made to `save_state()`, an `Underflow` exception will be thrown.

3.5.3 Exceptions of the `Function` Class

There are two exceptions.

- `class SanityError{};`

  Thrown in the usual circumstances.

- `class Underflow{};`

  Thrown if more calls are made to `restore_state()` than have been made to `save_state()`. Ths is only thrown if an internal error occurs.

3.5.4 Friends of the `Function` Class

The friends of the class are similarly few in number.

- `friend std::ostream& operator<< (std::ostream& s, const Function& f);`

  An output function.

3.5.5 Private Members of the `Function` Class

- `nat function_id;`

  The ID we obtained from `FunctionArguments::get_new_function_id()`.

- `nat arity;`

- `std::set<int> domain;`

  The function's arity and domain.

- `std::vector<const FunctionArguments*> default_ordering;`

  The default tuple ordering which was supplied to the constructor.

- `bool is_partial;`

  Is the function partial?

- `std::vector<std::pair<const FunctionArguments*, const FunctionValues*> > history;`

  A stack (actually, a `vector`) on which we store all changes made to the function after it has been created. A crucial part of the implementation of `restore_state()`.

  As an aside, note that if $f(\textit{some\_tuple})$ changes several times between successive calls to `save_state()`, then the restoration process will be unnecessarily long, since $f(\textit{that\_tuple})$ will change several times during restoration. Nevertheless, it is almost certainly faster to put up with this slight inefficiency than to try and keep track of which tuples of f have been changed since the last call to `save_state()`.

- `static std::pair<const FunctionArguments*, const FunctionValues*> history_separator;`

  Set up to be `<NULL, NULL>`, a `history_separator` is placed on the `history` stack every time a `save_state()` request is made. It allows `restore_state()` to know how many steps it has to restore.

3.6 The Constraint Class

3.6.1 Overview of the Constraint Class

The Constraint class contains no objects; rather it just defines a standard interface through which the constraint solver can communicate with individual constraints. The constraints themselves are all subclasses of the Constraint class, and will be considered separately below.

All Constraints lie within a ConstraintStore (see below) and are tailored to only restrict functions of a certain arity, and over a certain domain: an IdempotentConstraint on functions of arity three and domain size four is quite different to an IdempotentConstraint on functions of arity seven and domain size two. Individual constraints may have information on domain, arity, and several other variables passed to them when they are constructed, if this is desired.

Each constraint is called by the solver at least once at every search depth (until the constraint is entirely satisfied). The solver will pass the constraint a set of FunctionArguments whose associated FunctionValues have changed (shrunk in size) since the previous search level. It is then up to the constraint to see whether this new information can allow it to perform any further restrictions on the values taken by the function. If so, it restricts the function, and tells the solver which FunctionArguments it has changed.

The value which the constraint then returns to the solver can be one of several things. Firstly, it can be Unknown. This tells the solver that we have no idea whether or not the newly restricted function satisfies the constraint. Secondly, it can be NeverSatsified. This tells the solver that none of the functions described by the Function can satisfy the constraint. A third option is SometimesSatisfied, and a fourth is AlwaysSatisfied. These tell the solver that at least one, (resp. all) the functions defined by the Function satisfy the constraint. Finally, returning SymmetryBreaking is equivalent to returning Unknown or SometimesSatisfied (we do not make a distinction) for constraints whose only purpose is to restrict the number of functions output without changing the "essential" set of functions output. Examples of such constraints are the AutomorphismConstraint and the EssentiallyNAry constraint. Even these constraints, however, should return AlwaysSatisfied or NeverSatsified under the appropriate conditions.

The constraint lets the solver know about all the FunctionArguments it restricts via the is_constraint_relevant() function. When (and if) we reach a point in the search where all of these FunctionArguments take only one value apiece, the complete flag will be set to true when the apply_constraint() function is called. That is, the complete flag is set when (as far as the constraint is concerned) the function is completely defined.

A constraint which has not returned one of AlwaysSatisfied, SometimedSatisfied, or NeverSatisfied (and which restricts at least one FunctionArguments tuple) can be guaranteed to be called with the complete flag set to true, unless it is just a symmetry-breaking constraint and the user has selected a force-level of less than two.

When the complete flag is true, all the different return values for the constraint, apart from NeverSatisfied are interpreted in the same way: as saying that the function defined by the Function satisfies the constraint. In particular, a return value of Unknown is interpreted in this way.

The existence of the complete flag means that one simple way to write a new constraint is to write a routine which says:

```
if (!complete) return Unknown
// Writing the constraint is now much simpler
...
```

This should be avoided whenever possible, since it does not enable any pruning of the search tree to be done at a high level. In general, constraints should aim to be as ruthless at eliminating

FunctionValues at as high a search level as possible, and should also endeavour to return as descriptive a return-value as possible. Occasionally, speed constraints may prevent us from doing as much high-level pruning as we might like, but we should always aim to do as much as is practicable.

At each search depth, if any constraint modifies the FunctionValues of one or more FunctionArguments then these changes will be passed on to all constraints to see whether they can make use of these restrictions to restrict the FunctionValues of the function still further. Importantly, the modified FunctionArguments will also be passed back to the constraint which originally modified them. This may seem like a waste of time, but it is deliberate, and frequently helpful. It means that individual constraints do not have to worry about iterating themselves until they reach a stable state; instead, the solver will do this automatically, and call the constraint anew with its modified FunctionArguments.

Constraints typically do not know (and need not care) which search level the solver has reached: the above interface suffices in most cases. However, sometimes a constraint *does* need to know the current search depth: typically, this occurs when a constraint holds some private variables which need to be reset whenever a backtrack occurs. Important examples are the PolymorphismConstraints themselves. These constraints keep track of which rows of a relation have been eliminated at every search depth, so clearly knowledge of when a backtrack is about to occur is essential.

The Constraint class interface allows for this possibility. If a constraint advertises itself as needing "level-notification" then the solver will notify the constaint whenever it moves down or up the search tree (as well as when it begins and ends the search). This allows the constraint to keep its variables in sync with the search.

3.6.2 Public Interface to the Constraint Class

- `enum ConstraintType {Commutative, Idempotent, NearUnanimity, Automorphism, Polymorphism, Affine1, Affine2, Affine3, Conservative, TwoSemilattice1, Maltsev, EssentiallyNAry, Squashing, ZeroOneRange};`

  The different varieties of constraints which exist.

- `enum SatLevel {Unknown, SymmetryBreaking, NeverSatisfied, SometimesSatisfied, AlwaysSatisfied};`

  The different levels of satisfaction which a constraint can return.

- `enum Constrainedness {Unconstrained, SymmetryConstrained, Constrained};`

  Whether or not a constraint affects a FunctionArguments tuple. Unconstrained means *no*, Constrained means *yes*, and SymmetryConstrained means *yes, but only for the purposes of breaking symmetry.*

- `virtual ConstraintType get_constraint_type() const = 0;`

  Returns the type of the present constraint. In practice, this value is only used so that the solver can check to make sure that each constraint type other than Polymorphism only occurs once in the constraint store. The check is performed purely for the reason of efficiency.

- `virtual bool permutation_closed() const = 0;`

  Is the present constraint permutation-closed? That is, if f is a funcion which satisfies the present constraint, and g is a permutation of f, is g guaranteed to satisfy the constraint? Examples of constraints which are permutation-closed are the PolymorphismConstraints themselves, along with `IdempotentConstraints` (and may others). Examples of constraints which are not permutation-closed include the constraints which implement affine functions. The `AutomorphismConstraint` must not be configured to remove argument permutations when searching for affine functions.

*[*** A more sophisticated search for affine constraints might instead search for* permutations *of affine constraints. This would allow us to turn on the argument-permutation-removal when performing the search. ***]*

Under normal circumstances, permutation constraints are turned off automatically when non-permutation-closed constraints are turned on. If for any reason they are not, then a `ConstraintStore::ConsistencyError` exception will be thrown before the search begins.

- `virtual bool need_level_notification()const = 0;`

  A constraint should return true if it needs to be notified whenever the solver changes its search depth. As mentioned above, this is normally only required if a constraint has to maintain internal data structures.

- `virtual Constrainedness is_constraint_relevant(const FunctionArguments* fa) const = 0;`

  Does the constraint affect the `FuctionArgument` pointed to by `fa`?

- `virtual void check_consistency(nat a, const std::set<int>& dom, const std::set<const FunctionArguments*>& fas) const;`

  This function is called before the solver starts its search for polymorphisms. It gives the constraint a chance to carry out any sanity checks it wishes. For example, the `Commutative` constraint ensures that whenever $\langle x.y\rangle$ is a `FunctionArguments` tuple in `fas`, then $\langle y, x\rangle$ is also a member of `fas`. This saves an equivalent check having to be performed repeatedly whilst the search is underway.

  Minor warnings may be flagged with a message sent to `cerr`; major problems should cause a `SanityError` exception to be thrown.

- `virtual SatLevel apply_constraint(Function &f, const std::set<const FunctionArguments*> &modified_fas, std::set<const FunctionArguments*> &new_modified_fas, bool complete) const = 0;`

  This is the key function which is called by the solver repeatedly, and described in some detail above. On entry, `modified_fas` contains the set of `FunctionArguments` whose associated `FunctionValues*` have been modified since this function was last called. Any `Function-Arguments` whose values we in turn change should be added to the set `new_modified_fas` (even if they are already present in `modified_fas`).

  On entry to the function, `complete` will be `true` if the `Function f` is a completely specified function, (with respect to the `FunctionArguments` which the constraint restricts). (On rare occasions, `f` might be completely specified, but the solver hasn't yet noticed this fact, and `complete` is still `false`. In this situation however, the solver will later call every constraint with the `Function f` unchanged, but with `complete` set to `true`. So constraints which behave in the deprecated way described in the overview to this section will work without error. We stress again however, that this is not guaranteed to apply to symmetry-breaking constraints unless the user has selected a force-level of at least two.)

We return the satisfaction level (`AlwaysSatisfied`, `NeverSatisfied`, etc.) of the constraint, as described above. It is woth noting that we should return `NeverSatisfied` as soon as we notice that the constraint is violated: since we're about to backtrack, it doesn't matter if the data structures at the current level are left in a slightly inconsistent state.

- `virtual void display(std::ostream& s) const = 0;`

  Output the constraint in plain English.

- `virtual void notify_search_starting(nat max_d) {}`
- `virtual void notify_search_descent() {}`
- `virtual void notify_search_ascent() {}`
- `virtual void notify_search_ending() {}`

  A constraint which has indicated that it wishes to be notified about level changes should supply these four functions. The first is called just before the search begins. The maximum search depth which may be reached is supplied as an argument. This might be useful for resizing vectors, for example. The second function is called whenever we head deeper into the search tree; the third is called whenever we backtrack; and the fourth is called when the search finally ends. The current search depth is not passed as an argument: any constraint who wants to know what the current depth is should keep its own record.

  A constraint which has advertised itself as `AlwaysSatisfied` will not be called via these functions until the search depth recedes back to the level at which the constraint first became `AlwaysSatisfied`.

3.6.3 Exceptions of the `Constraint` Class

Just the one usual exception.

- `class SanityError{};`

3.6.4 Friends of the `Constraint` Class

Just the one usual friend.

- `friend std::ostream& operator<< (std::ostream &s, const Constraint &c)`

3.6.5 Protected Members of the `Constraint` Class

- `typedef std::vector<const FunctionArguments*>::const_iterator FAVI;`
- `typedef std::set<const FunctionArguments*>::const_iterator FASI;`

  A couple of handy `typedef`s.

- `bool restrict_range(const FunctionArguments* fa, int d, Function& f, std::set<const FunctionArguments*>& new_modified_fas) const;`

- `bool restrict_range(const FunctionArguments* fa, const FunctionValues* fv, Function& f, std::set<const FunctionArguments*>& new_modified_fas) const;`

  The functions are very handy utilities when it comes to implementing constraints. They restrict the range of `f[fa]` to being the singleton set {`d`}, and the set associated with `fv` respectively. If taking the intersection of the current set of values of `f[fa]` and the new set gives the empty set, then the function returns `true`,otherwise `false`. If the intersection is smaller than the original value of `f[fa]`, then `fa` is added to `new_modified_fas`.

  Put simply, this function allows the constraint to say something like "I know that $f(x, y, z) \in \{a, b, c\}$. Please update the function to reflect this, deal with the consequences, and return `true` if this leads to a contradiction."

- `bool restrict_to_equal_ranges(const FunctionArguments* fa1, const FunctionArguments* fa2, Function& f, std::set<const FunctionArguments*>& new_modified_fas) const;`

  This does a very similar job to the above two functions, but this time specifies that the ranges of `f[fa1]` and `f[fa2]` should be equal. It restricts the domains of both `FunctionArguments`, possibly adds both to `new_modified_fas`, and returns `true` if, and only if, this leads to a contradiction.

3.7 Subclasses of the `Constraint` Class

There are several varieties of constraints implemented. Since they all conform to the interface which has just been described, we omit it. We only describe those members which are unique to each constraint. Apart from the constructors, all members are private unless otherwise stated.

For further details of some of the algorithms employed by these constraints, see later in the manual.

3.7.1 The `PolymorphismConstraint` Class

Perhaps the most important class of constraints, these ensure that all functions generated are polymorphisms of the input relations. Each relation, R, generates several constraints, each of which says, in effect:

$$\begin{aligned}
f(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_a) &= R_1 \vee \\
f(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_a) &= R_2 \vee \\
&\vdots \\
f(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_a) &= R_s \vee
\end{aligned}$$

where a is the arity of functions we're searching for; s is the number of tuples in R; $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_a$ are tuples from R: constant for each constraint; and R_i represents the i^{th} tuple from R (according to some ordering scheme). As can be seen, each relation generates d^a constraints, where d is the domain size we're working over.

Each `PolymorphismConstraint` is initialised by its corresponding relation (see the member function `Relation::construct_constraints()`) using the following functions.

- `PolymorphismConstraint(const std::set<const FunctionArguments*>& fas, const std::set<int>& dom, nat arity);`

This constructor takes as input the set of `FunctionArguments` that the constraint restricts, as well as the domain and arity we're working over. The `FunctionArguments` we restrict correspond to the tuples

$$\begin{aligned}
&\langle \mathbf{t}_1[1], \mathbf{t}_2[1], \ldots, \mathbf{t}_a[1] \rangle \\
&\langle \mathbf{t}_1[2], \mathbf{t}_2[2], \ldots, \mathbf{t}_a[2] \rangle \\
&\vdots \\
&\langle \mathbf{t}_1[arity(R)], \mathbf{t}_2[arity(R)], \ldots, \mathbf{t}_a[arity(R)] \rangle.
\end{aligned}$$

Note that this constructor merely defines an empty constraint – it doesn't place any of the disjuncts necessary into the constraint. To do that, we need to call the function `add_possible_assignments()`.

- `~PolymorphismConstraint();`

  Destructor.

- `void add_possible_assignments(std::set<std::pair<const FunctionArguments*, int> > possibility);`

  This public function adds a disjunct to the constraint. The disjunct is specified by giving a mapping from each `FunctionArguments` tuple to the value it takes. If the map is inconsistent, (*e.g.,* we're trying to map $f(1, 2) = 3$ and $f(1, 2) = 1$ then the disjunct won't be added.

The remaining members are private.

- `std::set <const FunctionArguments*> restricted_fas;`

  Those `FunctionArguments` which we restrict.

- `std::map<const FunctionArguments*, std::vector<int> > possible_value_vectors;`

  The popssible values taken by each `FunctionArguments` tuple. The vectors are in the same order for each `FunctionArguments` – that is, each index into the vector gives one allowable simultaneous assignment to all the `FunctionArguments` in the map.

- `nat possible_value_vectors_size`

  The length of each vector in the above map. In theory, this is equal to the size of the relation which generated the constraint. In practice, because inconsistent maps aren't added, it may be less.

- `mutable std::list<nat> available_indices;`

  Those indexes into the vectors which have not yet been ruled out as impossible. It is always a sub"set" of $[0,$`possible_values_size`$)$.

- `mutable std::vector<std::list<nat> > unavailable_indices;`

  This `vector` contains, for every search depth, a list of indices which we removed from the above list of `available_indices` at that search depth. We need to keep track of this so that we can backtrack quickly by splicing the two lists together.

- Function* possible_value_function;

  This does exactly the same job as possible_value_vectors, above, but only keeps track of the *set* of values which can currently be taken by any FunctionArguments tuple. It is implemented as a function for speed. The reason for having two variables which encode essentially the same information is, as might be guessed, for speed: checking for set membership is fast, yet sets don't contain all the information present in a list.

- nat depth;

- nat max_depth;

  The current and maximum search depth (for use when we are notified about backtracking).

3.7.2 The AutomophismConstraint Class

This symmetry-breaking constraint forces functions which are identical up to permutations of their arguments, or up to the application of an automorphism to one or both sides of the function definition, or some sequence of these, to only be generated once.

- AutomorphismConstraint const std::set<const FunctionArguments*> &fas, const std::set<int>& dom, const RelationStore& relstore, bool argument_perms, bool da_left, bool da_right, bool da_both, bool accuracy = true);

  The constructor takes a lot of arguments. fas is a list of those FunctionArguments which the functions we generate will assign to. dom is the domain of the functions. relstore is the RelationStore whose automorphisms we should use to eliminate symmetries (it may be NULL if we're not interested in removing automorphism symmetries). The four bools indicate whether we are interested in removing symmetries due to argument permutations, applying automorphisms to the left of function definitions, to the right of function definitions, or to both sides of function definitions. Any number of these may be set to true (though if none of them are, the constraint won't do anything). Finally, accuracy should be set to true if speed should be sacrificed for accuracy. (If accuracy is false then this constraint will let through some functions which it would otherwise reject. It will never rejexct any functions which it would otherwise let through. So setting accuracy to false is safe. At the time of writing, accuracy is set to false by the calling program when using force levels of less than two.)

- bool argument_permutations_enabled() const;

- bool domain_automorphism_left_enabled() const;

- bool domain_automorphism_right_enabled() const;

- bool domain_automorphism_both_enabled() const;

  These four functions just tell the caller which of the four possible symmetries are currently enabled.

The remaining member are all private.

- std::set<int> domain;

  The domain we're working over.

- bool argument_perms;
- bool da_left;
- bool da_right;
- bool da_both;
- bool accuracy;

  Which symmetries are enabled, and are we going for accuracy over speed?

- typedef std::vector<const FunctionArguments*> FA_VEC;
- typedef std::map<int, int> DOMAIN_AUTOMORPHISM;
- typedef std::pair<FA_VEC, DOMAIN_AUTOMORPHISM> FN_GENERATOR;
- typedef std::set<FN_GENERATOR> ECG;

  A series of typedefs. An FA_VEC is just a vector of FunctionArguments. A DOMAIN_AUTOMORPHISM describes an injective unary function, as a map. A FN_GENERATOR describes how we have to manipulate a function so as to obtain a new function (typically, a new function in the same symmetry equivalence class as the original function). It essentially says "To create the characteristic tuple (see the Algorithms section, below) of the new function from f, put the FunctionArguments of f in the order given by FA_VEC, apply f to them to create a characteristic tuple of FunctionValues, then apply the DOMAIN_AUTOMORPHISM pointwise to each of these FunctionValues to get the new characteristic tuple."

  Finally, an ECG is just a set of FN_GENERATORs. It typically describes how to construct all the functions which are equivalent to f, from f.

- mutable bool done_ecg_initialisation;

  Have we set up the equivalence class of symmetries yet?

- mutable FA_VEC fa_ordering;

  The order we put FunctionArguments in to check for the lexicographically first characteristic tuple.

- mutable ECG ecg;

  The variable which actually contains the generators of the equivalence class.

- void initialise_ecg(const std::set<const FunctionArguments*> &fas, const RelationStore& relstore) const;

  Set up the ecg variable. If accuracy is false, then the resulting "equivalence class" will not necessarily be transitive(!). Consequently, if g and f are functions which are obtained from each other by permuting their arguments, or by applying an automorphism to their definitions, then only one will be output (assuming that all the flags apart from accuracy are set to true). But if g is obtained from f by permuting its arguments, *then* applying an automorphism, then g and f will both be output. If accuracy is set to true then transitivity is enforced, and once again only one of f and g will be output.

  The FunctionArguments and RelationStore arguments should be the same as those in the constructor. *[*** We don't currently store them in the class, since this function is currently* *called directly from the constructor, and there is no point. Future versions may call this function later on, in which case these will probably not be passed as arguments, but will be stored away. ***]*

- `void compose_automorphisms(const DOMAIN_AUTOMORPHISM& a1, const DOMAIN_AUTOMORPHISM& a2, DOMAIN_AUTOMORPHISM& result) const;`

  Composes the two supplied automorphisms. Application is on the left, that is `result`$(x) =$ `a1`(`a2`(x)).

  This is not a `static` function, though it could be. The reason it isn't is because we can do sanity checks to make sure that the domains of `a1` and `a2` are consistent with what we think they ought to be.

- `void apply_automorphism_to_fa_vec(const DOMAIN_AUTOMORPHISM& da, const FA_VEC& fa_vec, FA_VEC& result) const;`

  Applies the given automorphism pointwise to each `FunctionArguments` in the vector `fa_vec` (or rather, to their corresponding tuples). The result is stored in `result`.

- `void add_fn_gens_argument_perms(const ECG fn_gens, ECG& additional_fn_gens) const;`

- `void add_fn_gens_da_left(const ECG& fn_gens, const std::set<DOMAIN_AUTOMORPHISM>& iufs, ECG& additional_fn_gens) const;`

- `void add_fn_gens_da_right(const ECG& fn_gens, const std::set<DOMAIN_AUTOMORPHISM>& iufs, ECG& additional_fn_gens) const;`

- `void add_fn_gens_da_both(const ECG& fn_gens const std::set<DOMAIN_AUTOMORPHISM>& iufs, ECG& additional_fn_gens) const;`

  Each of these functions (one for each of the four types of symmetry this constraint removes) does essentially the same job, and it's basically the same job that is done by the members of the `Constraint` class itself. They take as input a set of `FN_GENERATORS`, and work out what new `FN_GENERATORS` must be added to the set to close it under their symmetry. These additional generators are placed in the set `additional_fn_gens` on exit. The three symmetries which rely on knowledge of the injective unary polymophisms of the `RelationStore` which was supplied to the constructor of this constraint, are passed the list of such functions (in the form of `map`) as part of their input.

3.7.3 The `EssentiallyNAryConstraint` Class

This constraint ensures that any polymorphisms output use all of their arguments in an essential way. It is a symmetry-breaking constraint, so if the user supplies a `force` option of less than 2, some functions may be output which do not, in fact, make essential use of all their arguments. Indeed, since the implementation makes use of the `complete` flag (described above as deprecated) then it can sometimes happen that a function is completely described, does not use all of its arguments, yet is still not removed by this constraint at force levels of less than two.

- `EssentiallyNAryConstraint(const std::set<const FunctionArguments*>& fas, const std::set<int>& dom, nat a);`

Constructor. Supply set of those `FunctionArguments` which define the arguments of the functions we're searching for, along with the domain and arity of the functions we're searching for. (The first of these could, of course, be deduced from the latter two, but it's easier not to have to calculate the set twice.)

The remaining functions and variables are `private`.

- `std::map<const FunctionArguments*,`
  `std::vector<std::list<constFunctionArguments*> > > modifications;`

  `modifications[fa][i]` contains a list of those `FunctionArgument*`s which can be obtained from the `FunctionArgument*` by replacing the $\mathtt{i}^{\mathrm{th}}$ element of its corresponding tuple with a different value from `domain`.

- `mutable std::list<nat> undetermined_fn_posns;`

  A list of indexes (from 0 up to `arity` -1) which have not yet had essential use made of them. This list begins by containing every possible index; when the list becomes empty then we know that the constraint is `AlwaysSatisfiable`.

- `mutable std::vector<std::list<nat> > essential_fn_posns;`

  For each search depth, a list of indexes which we discovered at that depth to be used in an essential way.

- `nat arity;`

- `std::set<int> domain;`

  The arity and domain of the functions we are searching for.

- `nat depth;`

- `nat max_depth;`

  The current and maximum search depths.

3.7.4 The `AssociativeConstraint` Class

This constraint may only be applied when we are searching for arity-2 functions. It ensures that they are associative. The algorithm we currently use to check for associativity could probably be improved. See the source code for more details.

- `AssociativeConstraint(nat a, const std::set<int>& dom);`

  The constructor takes as arguments the arity of polymorphism we are searching for, and the domain over which the search takes place.

- `nat arity;`

- `std::set<int> domain;`

  It saves them in these private variables.

3.7.5 The `CommutativeConstraint` Class

This constraint may only be applied when we're searching for arity-2 functions. It ensures that they are commutative. Its constructor takes no arguments.

3.7.6 The `IdempotentConstraint` Class

Forces functions to be idempotent.

- `IdempotentConstraint(nat a);`

  This constructor must be passed the arity of functions we're searching for as an argument.

- `nat arity;`

  It is squirrelled away by here.

3.7.7 The `ZeroOneRangeConstraint` Class

This is a slightly unusual constraint. It simply ensures that all `FunctionValues` are subsets of $\{0, 1\}$, and that the `FunctionValues` of constant `FunctionArguments` are just the set $\{0\}$. This is unusual, because there is normally no guarantee that 0 and 1 will be in the domain of function we're searching for. (If they are not, a `SanityError` will be thrown at the consistency-checking stage.) This constraint is probably only useful for the purpose it was originally written for: as part of the code which enumerates all essentially different relations. It essentially foces all polymorphisms to be charactersistic functions of non-trivial relations.

3.7.8 The `NearUnanimityConstraint` Class

Forces functions (of arity at least three) to be near-unanimity functions.

- `NearUnanimityConstraint(nat a)`

  This constructor must be passed the arity of functions we're searching for as an argument.

- `nat arity;`

  It is squirrelled away by here.

3.7.9 The `Affine1Constraint` Class

This can only be applied when we are searching for arity-3 functions. It implements the constraint that $f(f(x, y, z), z, u) = f(x, y, z)$.

- `Affine1Constraint(nat a, const std::set<int>& dom);`

  This constructor must be passed the arity and domain of functions we are searching for.

- `nat arity;`

- `std::set<int> domain;`

  They are stored here.

3.7.10 The `Affine2Constraint` Class

This can only be applied when we are searching for arity-3 functions. It implements the constraint that $f(x, y, z) = f(z, y, x)$. Its constructor takes no arguments.

3.7.11 The `Affine3Constraint` Class

This can only be applied when we are searching for arity-3 functions. It implements the constraint that $f(x, x, y) = y$. Its constructor takes no arguments.

Between them the `Affine1`, `Affine2`, and `Affine3` constraints ensure that a function is affine.

3.7.12 The `ConservativeConstraint` Class

This implements the constraint that $f(x_1, x_2, \ldots, x_n) \in \{x_1, x_2, \ldots, x_n\}$. Its constructor takes the function arity as its sole parameter:

- `ConservativeConstraint(nat a);`
- `nat arity;`

  The arity is stored away in the private `arity` variable.

3.7.13 The `TwoSemilattice1Constraint` Class

This constraint may only be applied when we are searching for arity-2 functions. It implements the constraint $f(x, f(y, x)) = f(y, x)$.

Together with the `CommutativeConstraint` and `IdempotentConstraint` classes, this constraint restricts arity-2 functions to being 2-semilattices.

- `TwoSemilattice1Constraint(nat a, const std::set<int>& dom);`

  This constructor must be passed the arity and domain of functions we are searching for.
- `nat arity;`
- `std::set<int> domain;`

  They are stored here.

3.7.14 The `MaltsevConstraint` Class

This constraint may only be applied when we are searching for arity-3 functions. It implements the constraint $f(x, y, y) = f(y, y, x) = x$. Its constructor takes no arguments.

3.7.15 The `SquashingConstraint` Class

This constraint can only be applied when we are searching for arity-1 functions. It implements the constraint that f is non-injective.

- `SquashingConstraint(const std::set<int>& dom);`

  The constructor must be passed the domain of functions we're searching for.
- `std::set<int> domain;`

  It is stored away here.

3.8 The SolutionReceiver Class

This very simple class merely defines two `typedef`s and (for safety) a virtual destructor. All are `public`.

- `virtual ~SolutionReceiver(){}`
- `typedef SolutionReceiver* CallbackObject;`
- `typedef void (SolutionReceiver::*CallbackFunction)(const Function&);`

Any function which wishes to receive notification of each solution found by the solver, must be declared as
`void the_function(const Function&)`
and be a member function of an object which is a member of the `SolutionExists` class. A pointer to the object, and a member-pointer to the function must then be passed to the function `ConstraintStore::find_polymorphisms()` when it is first invoked.

It may be necessary to cast the pointer and member-pointer to be of the appropriate type. This may be done using `(CallbackObject) obj_pointer` and `(CallbackFunction) mbr_pointer` respectively.

3.9 The ConstraintStore Class

3.9.1 Overview of the ConstraintStore Class

There is little to say about the `ConstraintStore` class which has not been said already. Each `ConstraintStore` object comes equipped with an arity and a domain, it contains several constraints on functions of that arity and over that domain, and it incorporates a polymorphism-finding constraint solver to find all the functions which satisfy the constraints.

3.9.2 Public Interface to the ConstraintStore Class

- `ConstraintStore(nat a, const std::set<int>& dom);`

  The constructor. Supply the arity and domain of functions which the solver should look for.

- `~ConstraintStore();`

  Destructor. *This deletes any constraints within the store.* If you don't want some of the constraints to be deleted, you will have to take a copy of them.

- `bool remove_constraint(Constraint* c, bool del = true);`

  Removes a constraint from the store. If `del` is set to `true` on entry, then the constraint will be deleted. Returns `false` under normal circumstances, or `true` if the constraint was not present in the store.

- `nat get_arity();`

  Returns the arity of the functions we generate.

- `void add_constraint(Constraint* c);`

  Add the constraint pointed to by `c` to the store.

- `void display(std::ostream& s) const;`

  Display all the constraints, in plain English, on stream `s`.

- `nat find_polymorphisms(bool quiet, bool stop_at_first, bool force_complete_functions, bool force_all_symmetry_constraints, bool force_essential_constraints, SolutionReceiver::CallbackObject co = NULL, SolutionReceiver::CallbackFunction cf = NULL);`

  Generate all the polymorphisms. The Boolean arguments correspond to a subset of the options detailed in the `UserOptions` class. For more details see the description of that class. Briefly however, `quiet` prevents functions from being displayed on screen as they are found; `stop_at_first` stops (and throws a `SolutionExists` exception) as soon as it encounters its first solution; `force_complete_functions` forces all functions output to be complete, in the sense that every `Function` output will correspond to a unique mathematical function; `force_all_symmetry_constraints` forces all functions output to obey all the symmetry constraints; and `force_essential_constraints` forces all functions output to obey all the non-symmetry constraints.

  Whenever a solution is found, the supplied member function `co.cf` will be called, and supplied with a description of the solution found. See the `SolutionReceiver` class for a description of what the declared type of the member function should be. If `co` is `NULL`, then no function will be called when a solution is found.

  We return the number of solution specifications we found.

- `AssociativeConstraint* add_associative_constraint();`

- `CommutativeConstraint* add_commutative_constraint();`

- `IdempotentConstraint* add_idempotent_constraint();`

- `NearUnanimityConstraint* add_near_unanimity_constraint();`

- `EssentiallyNAryConstraint* add_essentiallynary_constraint();`

- `void add_affine_constraint();`

- `ConservativeConstraint* add_conservative_constraint();`

- `MaltsevConstraint* add_maltsev_constraint();`

- `SquashingConstraint* add_squashing_constraint();`

- `void add_two_semilattice_constraint();`

- `ZeroOneRangeConstraint* add_zero_one_range_constraint();`

- `PolymorphismConstraint* add_polymorphism_constraint(const std::set<const FunctionArguments*>& fas);`

- `AutomorphismConstraint* add_automorphism_constraint(const RelationStore& relstore, bool argument_perms, bool da_left, bool da_right, bool da_both, bool accuracy = true);`

  These are just short-cuts for adding certain types of constraints to the store. They each create a new constraint of the appropriate type, add it to the store, and (mostly) return a pointer to it. For the `PolymorphismConstraint`, the argument supplied tells us which `FunctionArguments` the constraint restricts (since the constructor for that constraint requires that information). For the `AutomorphismConstraint`, we need to know which (if any) of the four symmetries we want to eliminate, along with the `RelationStore` which we use to find the unary automorphisms (or `NULL` if desired, and we're only eliminating argument permutations). Finally, we also need to specify whether we should value accuracy over speed (the default).

The `void` functions are those which actually involve adding several constraints to the store. We could return a set of pointers, but instead we return nothing.

3.9.3 Exceptions of the `ConstraintStore` Class

- `class SanityError{};`

  The usual sanity error.

- ```
  class SolutionExists
  {
  public:
  SolutionExists(const Function& f);
  SolutionExists(const SolutionExists& copy_from);
  ~SolutionExists();
  Function* the_solution; };
  ```

  Thrown whenever `stop_at_first` is set to `true`, and we find a solution. The solution should be passed to the constructor, and is stored in the public variable `the_solution`.

- `class ConsistencyError;`

  We have been asked to do something contradictory: find all arity-3 commutative functions, for example.

3.9.4 Friends of the `ConstraintStore` Class

Just the usual output function.

- `friend std::ostream& operator<< (std::ostream &s, const ConstraintStore &cs)`

3.9.5 Private Members of the `ConstraintStore` Class

The members in the following list form just a subset of the private members of the `ConstrainStore` class. There are a dozen or more other private members in the code which are for exclusive use of the constraint-solving code. Normally, these would be local to the solving routines, but for the sake of speed, I have made them global to the `ConstraintStore` object. (I did try making them truly global, but the speed increase was negligable.) All these variables and member functions are well-commented in the code, but it should not be necessary to modify them unless the constraint solver itself is modified. The relative ugliness of the solver is buried in a source file of its own: `solver.cc`.

- `std::set <Constraint*> constraints;`

  The set of (pointers to) constraints in the store. This is a superset of:

- `std::set <Constraint*> notifiable_constraints;`

  The set of constraints in the store which need to be notified when the search depth changes. *[*** Come to think of it, I can only think of one place in the code which uses this variable; and that uses it inessentially. We could probably delete it. ***]*
  ```

- typedef std::set<Constraint*>::const_iterator CSI;

  The type of iterators over ConstraintStore members.

- std::set<const FunctionArguments*> constrained_fas;

  All the FunctionArguments we're constraining. That is, all arity-tuples of elements from domain.

- std::vector<const FunctionArguments*> lex_ordering;

  The same, in lexicographic order. (Of course, the reason for having both variables is that we sometimes need to be able to check whether we're constraining a particular FunctionArguments tuple.)

- nat arity;

  The arity of functions we'll be searching for.

- std::set<int> domain;

  And the domain.

3.10 The UserOptions Class

3.10.1 Overview of the UserOptions Class

An object of the UserOptions class parses the command line supplied by the user, and stores away the various options they request. It can then be quizzed to find out the values of these options. There is very nearly (but not quite) a one-to-one correspondence between the options available to the user, and the variables we store.

3.10.2 Public Interface to the UserOptions Class

There is only one substantive function supplied by the UserOptions class. Almost all of the others are trivial, one-line functions.

- UserOptions();

  Constructor. Sets up the default values of the options.

- nat read_options(nat argc, char* argv[]);

  Passed the variables argc and argv which are handed to the main() function when the program begins, this function parses the command line, and stores away the values of the options in some private variables. It returns the array index of the first filename on the command line (or argc if there does not appear to be a filename.)

- void display_short_help() const;

  Displays a help message, for display when the user runs Polyanna with no arguments. Just a short welcome message.

- void display_long_help() const;

  Displays a help message, for display when user runs Polyanna with the --help switch. Displays brief help information, along with a listing of all command-line options.

- `nat get_fn_arity() const;`

  The arity of polymorphism we are looking for.

- `bool get_perm_opts() const;`

  Do we want to turn on the permutation optimisations?

- `bool get_automorphisms() const;`

  Do we want to factor out automorphisms?

- `bool get_essentially_nary() const();`

  Do we want every function listed to use all of its arguemnts essentially?

- `bool get_commutative() const;`

  Do we just want to search for commutative functions? (Arity 2 only.)

- `bool get_idempotent() const;`

  Do we just want to search for idempotent functions?

- `bool get_near_unanimity() const;`

  Do we just want to search for near-unanimity functions?

- `bool get_affine() const;`

  Do we want to just search for affine functions?

- `bool get_conservative() const;`

  Do we just wish to search for conservative functions?

- `bool get_libdem() const;`

  Do we just wish to search for Liberal Democratic functions?

- `bool get_twosemilattice() const;`

  Do we just wish to search for 2-semilattice functions?

- `bool get_maltsev() const;`

  Do we just wish to search for Mal'tsev functions?

- `bool get_squash() const;`

  Shall we try to squash the domain of the input relations before we begin the search?

- `bool get_force_essential_constraints() const;`

  If `true`, all functions which are output must satisfy all the non-symmetry-breaking constraints.

- `bool get_force_all_symmetry_constraints() const;`

  If `true`, all functions which are output must satisfy all the symmetry-breaking constraints.

- `bool get_force_complete_functions() const;`

  If `true`, all functions which are output must be "complete" – that is, each `FunctionValues` of the function must represent a set of size exactly one.

- `nat get_force_level() const;`

  Return the force level (0 to 3) as an integer. Strictly speaking, this function is unnecessary given the existence of the above three functions (and vice versa). But it's sometimes nice to have both available.

- `bool get_stop_at_first() const;`

  Stop, and throw a `SolutionExists` exception, as soon as we spot our first solution. The exception contains details of the solution found.

- `bool get_quiet() const;`

  Suppress the displaying of solutions?

- `bool get_count() const;`

  Do we display a final count of the number of solution specifications found?

- `bool get_tractable() const;`

  If `true`, we are only interested in knowing whether or not the input relations form a tractable set. We are not interested in knowing what the polymorphisms are, *per se*.

- `bool get_view_constraints() const;`

  If `true`, we display the constraints present in the store, before beginning the search.

- `bool get_view_relations() const;`

  If `true`, we display the decomposition of all the input relations, before commencing the search.

- `nat get_iterations() const;`

  The number of times we should iterate the search for solutions. This will be 1 under almost all circumstances, but may be set to a higher value during debugging, when we want to know how long it takes to execute the solving routine, say, 5000 times. It may also be set to 0 if the user asks for the constraint solver not to be invoked.

- `int get_enumerate_relations() const;`

  Do we want to behave as a relation enumerator? -1 for "no", a non-negative integer n for "yes, over a domain of size n".

- `void display_options_summary() const;`

  Displays a brief summary of the most important options selected. This is typically called by the main program after it has displayed the title, and before beginning the polymorphism search.

3.10.3 Exceptions of the `UserOptions` Class

- `class SanityError;`

  The usual sanity error exception.

- ```
  class UnknownOption
  {
  public:
  UnknownOption(std::string s)
  {option = s;}
  std::string option;
  };
  ```
  An unrecognised command-line option was passed. The `string s` contains the option we are having difficulty with.

- ```
  class MalformedOption
  {
  public:
  MalformedOption(std::string s)
  {option = s;}
  std::string option;
  };
  ```
  Although we recognised the option, we are confused by some arguments which were passed to it. The `string s` contains the option we are having difficulty with.

3.10.4 Private Members of the `UserOptions` Class

The only private members of the `UserOptions` Class are variables which hold precisely the values returned by the routines already described in the `public` section. They are omitted for brevity.

3.11 The `RelationEnumerator` Class

3.11.1 Overview of the `RelationEnumerator` Class

The `RelationEnumerator` class is a very simple class, which is just an object-ified interface to the function which enumerates all non-trivial, essentially different relations (over a domain of a given size, and of a given arity). It is a subclass of the `SolutionReceiver` class.

3.11.2 Public Interface to the `RelationEnumerator` Class

- `RelationEnumerator(nat arity, nat domain_size);`

  Constructor. Supply the arity and domain size of the relations we're interested in finding.

- `nat generate_relations(bool quiet = false);`

  Actually generate the relations, and display them on `cout`.

3.11.3 Private Members of the `RelationEnumerator` Class

- `nat arity;`
- `nat domain_size;`

  The arity and domain size we're looking for.

- `void polymorphism_receiver(const Function& f) const;`

  The function whose presence is mandated by the `SolutionReceiver` class. It receives the polymorphisms, and uses the `Function::display_as_relation()` function to display them.

3.12 Non-Member Functions

In this section, we detail those functions which aren't a member of any class. Most of them are general-purpose utility functions, which might be called from anywhere within the program.

- typedef unsigned int nat;

  Not a function. Just a typedef to save on typing. The type of natural numbers. We've seen this used throughout the program.

- int highest_common_factor(int x, int y);

  Returns the highest commmon factor of the integers x and y. The function uses Euclid's algorithm.

- bool is_subset(const std::set<int> &s, const std::set<int> &t);

  Returns true if, and only if, s is a subset of t.

- inline int min_set_value(const std::set<int>& s);

- inline int max_set_value(const std::set<int>& s);

  These return the smallest (and largest, respectively) values which are present in the set s. Their return value is undefined if the set is empty.

- void display_tuple(std::ostream &s, const std::vector<int> &t);

  Output a vector on stream s.

- void display_set(std::ostream &s, const std::set<int> &c);

  Output a set on stream s.

4 Algorithms

This section is devoted to describing the various non-trivial algorithms which are used in the program.

4.1 Decomposition of Relations

This section describes the algorithms we use when decomposing relations into simpler forms. Throughout, we treat relations interchangably both as sets of tuples, and as matrices. When viewing a relation as a matrix, the rows correspond to the tuples from the relation (in some fixed order).

The relation being analysed is referred to as R; its arity and size are a and s respectively.

For the sake of completeness we describe all decomposition functions, rather than just those with non-trivial algorithms.

4.1.1 Trivial Relations

The function check_for_trivial_relations() is itself trivial. It merely checks to see whether the present relation has an arity or size of 0. If so, then the relation is trivial.

4.1.2 Duplicated Columns

Again, this is a simple function. We can do slightly better than the $O(n^2)$ naïve algorithm however, by making use of the `set` container present in C++'s standard library.

> $SET\_OF\_COLUMNS = \emptyset$
> **for** each column $col \in R$
> $SET\_OF\_COLUMNS := SET\_OF\_COLUMNS \cup \{col\}$
>   **if** $SET\_OF\_COLUMNS$ has not changed **then**
>   Mark column col for deletion.

Finally, `delete_columns()` is called to perform the actual deletion. Depending on the implementation of sets, this is up to a times faster than the naïve algorithm. (In the case of C++, it is around $a/\log a$ times faster.)

4.1.3 Allows Anything

As indicated above, an `AllowsAnything` relation is, by definition, an arity-1 relation whose real and apparent domains are the same. This is trivial to check for.

4.1.4 Cartesian Powers

A `CartesianPower` relation can be characterised by the property that its apparent domain size, when raised to the power of a (its arity), is equal to s (the number of tuples in the relation). This property is the basis of the test in `check_for_cartesian_power()`.

4.1.5 Cartesian Products

The check to determine whether R is the Cartesian Product of two other relations has several stages. Firstly, if R is of size 1 then the relation can be immediately split into a arity-1, size-1 relations. (It is not essential that this is treated as a special case, but it is more efficient to do so.)

If R is the Cartesian product of two relations, R_1 and R_2, then the original relations can be recovered by projecting R onto the columns which originally came from R_1 and the columns which originally came from R_2. Before we can do this however, we need to know which columns came from which original relation. In the case of a straightforward product – $R_1 \times R_2$ – there is little problem: there are only $a - 1$ possible partition points to try. So the first stage of our algorithm tests these natural partition points. To do this effectively we need to be able to test whether any given split really does partition R into two sub-relations whose Cartesian product is R. This is easy; we shall come to the details in a moment.

If none of the natural parition points succeed in breaking up R, then things are a little harder. It is quite possible that R is a Cartesian product, but one in which the columns of $R_1 \times R_2$ have been permuted. How do we find the correct partition of R into two sets of columns?

The next stage of our algorithm provides a heuristic which can solve this problem in many cases. It first involves calculating an *hcf* value for each column of the relation, as follows.

> **for** each column, col
> $SET\_OF\_OCCURRENCES = \emptyset$
>   **for** each domain element $d \in col$
>   Count the number of occurrences of d in col. Call it n.
>   **if** $n > 0$

$$SET\_OF\_OCCURRENCES := SET\_OF\_OCCURRENCES \cup \{n\}$$
$$hcf[col] := \textit{Highest Common Factor}\ (SET\_OF\_OCCURENCES)$$

If any of the $hcf[]$ values are equal to the size, s, of the relation itself, then the associated columns must contain a constant value. These columns can be split off as size-1, arity-1 relations. We don't (as it happens) continue to look for other, more subtle Cartesian products in this case. They will be found when the relation's children are later examined with a view to decomposing them.

If any of the $hcf[]$ values are 1, then we know that R is certainly not a Cartesian product.

If all the $hcf[]$ vales are 2 or more, we then try splitting the relation in the following way. The first relation is generated from all those columns col, for which the highest common factor of $hcf[1]$ and $hcf[col]$ is greater than 1. The second relation is generated from all the remaining columns. We then use the test mentioned above to determine whether the Cartesian product of these two relations is R itself.

If the above tests all fail, then we resort to the brute-force approach of trying every one of the 2^{a-1} possible partitions to see whether it generates a successful Cartesian product decomposition. Because this may take a very long time for relations of a high arity, we only attempt this for relations of arity 10 or less.

Finally, we come to the problem we alluded to above: we need a test which, given a putative partition, can determine whether or not it does indeed define two relations whose Cartesian product is R. The function `Relation::check_split_for_cartesian_product()` implements the following test. First, it generates the two smaller relations induced by the partition. It then takes the product of their sizes. The partition is a true partition if, and only if, this product is equal to the size, s, of R itself.

4.1.6 Disjoint Domains

Determining whether or not a relation can be decomposed into the union of two relations over disjoint domains is not quite as easy as might be thought at first sight. First, we have to calculate what the disjoint domains actually are, and this is not trivial. For example, the relation $\{\langle 1,2\rangle, \langle 3,4\rangle, \langle 4,1\rangle\}$ over the domain $\{1,2,3,4\}$ cannot be split up, but we need to examine all three tuples before we can determine this. In general, splitting a relation into the disjoint union of two other relations is equivalent to finding the connected components of a graph (specifically, the Gaifman graph induced by the hypergraph of the relation). In the case of the above relation, the graph would contain an edge between vertex 1 and vertex 2, another between vertex 3 and vertex 4, and another between vertex 4 and vertex 1. This graph contains only one connected component, corresponding to the fact that the relation cannot be decomposed into the disjoint union of two smaller relations.

The following pseudo-code describes the an adapted connected-component-finding algorithm. It is used in `Relation::check_for_disjoint_domains()`. The idea is to associate a value with each tuple in the relation, so that any tuples which have a domain element in common share the same value. To aid in exposition, the pseudo-code assumes that these values are themselves tuples; the actual program code uses integers instead. Also, the code relies on the fact that tuples can be ordered in some manner (lexicographically, in practice), and that a **for** loop will visit the tuples of R in order. So if t is a tuple from R, it makes sense to say "**for** each tuple $t_2 < t$".

for each tuple $t \in R$
    $TUPLE\_VALUE[t] := t$
    for each tuple $t_2 < t$ *(Tuples we have already examined)*
        if $(TUPLE\_VALUE[t] = TUPLE\_VALUE[t_2])$

continue *(Try next t_2)*
if t and t_2 have a domain element in common
$min\_val := min\{TUPLE\_VALUE[t], TUPLE\_VALUE[t_2]\}$
$max\_val := max\{TUPLE\_VALUE[t], TUPLE\_VALUE[t_2]\}$
$TUPLE\_VALUE[t] := min\_val$
$TUPLE\_VALUE[t_2] := min\_val$
for each tuple $t_3 < t$
if $(TUPLE\_VALUE[t_3] = max\_val)$
$TUPLE\_VALUE[t_3] := min\_val$

The algorithm works as follows. We work through the tuples of R one by one. Each tuple is initially given a unique value (in practice, itself). We then check all previous tuples we've visited to see whether any of them overlap with the present tuple. If so, we make sure that both tuples (t and t_2) have the same value (min_val) as each other, and also make sure that any *other* tuple, t_3, which had the same value as either t or t_2 (that is, min_val or max_val), is also given the value min_val.

Having performed the above algorithm, it is trivial to break R up into the disjoint union of two smaller relations.

One final note. Although arity-1 relations are trivially decomposible into s arity-1, size-1 relations, this is a simplification we never make. The reason is simply that it is a useless simplification. We only search for disjoint unions when $a \geq 2$.

4.2 Implementing Constraints

We have already talked in a certain amount of detail about how many of the constraints are implemented. Most are straightforward, at least in theory (though an efficient implementation is often slightly hairy). This section is restricted to explaining, at a fairly high level, some of the less obvious algorithms used.

4.2.1 The `AutomorphismConstraint`

The `AutomorphismConstraint`, as we have seen, is a symmetry-breaking constraint which ensures that functions are only listed once up to permutation of their arguments. The idea behind its implementation is quite easy. To simplify exposition, we will restrict our attention to eliminating functions which are equivalent up to permutation of their arguments. The general method of eliminating automorphisms is essentially identical. See the `AutomorphismConstraint` itself, above, for more details.

Let D be some finite domain, and $k \in N$ some arity. We may fix some ordering, μ, of the members of D^k (lexicographic order, perhaps). Any function $f : D^k \longrightarrow D$ then gives rise to a $|D|^k$-tuple of elements of D which gives the value of f on every possible tuple of arguments. If $g \neq f$ is a function which is obtained from f by permuting its arguments, then g will also give rise to a such a *characteristic tuple*, which is different to the characteristic tuple of f.

The basic algorithm employed by the `AutomorphismConstraint` should now be fairly clear. Whenever a function is generated by the constraint solver, we examine the characteristic tuple of each of its permutations. If the present function gives rise to the minimal characteristic tuple, then we output the function; if not, we silently backtrack.

An example might make things clearer.

Example 4.2.1.1 *Let $D = \{0, 1\}$, and let $f : D^2 \longrightarrow D$ be the arity-2 function which simply projects onto its first argument. Then the characteristic tuple of f (according to the lexicographic* *ordering on D^2) is $\langle 0,0,1,1\rangle$. There is exactly one permutation of f: the arity-2 function g which projects onto its second argument. The characteristic tuple of g is $\langle 0,1,0,1\rangle$, which is lexicographically greater than the characteristic tuple of f. In this instance therefore, it is f which we output, and g which we ignore.*

So much for the theory. In practice the algorithm is far from ideal as stated, because it relies on a function specification being complete before we can determine whether or not it ought to be ignored. We can do a lot better than this. Suppose that D is the set $\{0,1,2,3\}$, that the characteristic tuple specification of the function specification f is $\langle\{1\},\{2,3\},\ldots\}\rangle$ and that g is a permutation of f whose characteristic tuple specification is $\{\{1\},\{0,1\},\ldots\}$. Then regardless of the portions of the characteristic tuple specifications which we have ellided with ellipses, we can deduce that none of the functions induced by f can be the lexicographically first of their permutations, and so we can backtrack immediately.

Another issue which arises from the algorithm as stated is the tuple-ordering μ which defines how the characteristic tuples are constructed. Different orderings will give rise to different function permutations being viewed as canonical, and may also affect the speed of program execution: if the search algorithm assigns, say, $f(0,1)$ and $f(1,0)$ very high up in the search tree, it would be advantageous for $\langle 0,1\rangle$ and $\langle 1,0\rangle$ to appear early on in μ to enable permutation-backtracking to occur as early as possible. On the other hand, if $f(0,1)$ was assigned very early, and $f(1,0)$ was assigned very late, then $\langle 0,1\rangle$ and $\langle 1,0\rangle$ ought probably to be left until quite on in μ.

This situation is complicated still further by the fact that Polyanna uses dynamic variable-ordering. It is quite possible for $f(0,1)$ to be assigned a value very early on in one branch of the search tree, but very late in a different branch.

As the program stands, the ordering μ is fixed, but it is not lexicographic. First of all, constant-tuples are omitted from μ, since any permutation of a function f will leave the value of each $f(\textit{constant\_tuple})$ unchanged. Secondly, tuples which are permutations of each other (such as $\langle 0,1\rangle$ and $\langle 1,0\rangle$) occur together as "blocks" in μ. Finally, these blocks are ordered so that the lexicographically first tuples of each block occur within μ in lexicographic order.

As a heuristic, I have given some thought to replacing the third of these conditions with the condition "These blocks are ordered so that the lexicographically first tuples of each block occur within μ in an ordering defined by μ'." The ordering μ' can be several things: one alternative which seems promising (but which I have not implemented) would be to disable the permuation checking constraint until we find our first solution, then set μ' to be the order in which we assigned values to the tuples. This heuristic may be effective if the order in which tuples are assigned values is broadly similar in each branch of the search tree.

4.2.2 The `EssentiallyNAryConstraint`

This constraint endeavours to ensure that every polymorphism output makes essential use of all of its arguments. It is a symmetry-breaking constraint, and so if the user has specified a force level of less than two, there is no guarantee that all functions output will satisfy this constraint. This is probably just as well, since it is the least satisfyingly implemented of all the constraints, and the only major one which currently makes use of the deprecated `complete` flag (see section 3.6.1). (The `SquashingConstraint` also makes use of the `complete` flag, but in that case it is impossible to know, even in principle, that a function is injective until it is complete.)

The primary difficulty in implementing this constraint is that it is very difficult to tell in advance if a function is going to ignore one of its arguments. Consider the *and* function on a Boolean domain, for example. We might have determined that $f(0,0) = 0$, $f(0,1) = 0$, and $f(1,0) = 0$, but until we find out the value of $f(1,1)$, we cannot know whether f is essentially nullary or essentially binary. The same problem appears even more acutely if we consider the generalisation of *and* to functions of arity 3 or greater.

In general, it is normally impossible to tell that a function is going to ignore one of its arguments until we are extremely close to the bottom of the search tree; Polyanna does not even try. Instead, it keeps track of which arguments have not yet been used essentially; if this set ever becomes empty then the constraint returns `AlwaysSatisfied`; if is is non-empty when we reach the bottom of the search tree (and `complete` is set to `true`), it returns `NeverSatisfied`. Otherwise, it just returns `SymmetryBreaking`.

Since a symmetry-breaking constraint is not guaranteed to be called with `complete` set to `true` unless the user has selected a force level of at least 2, this constraint occasionally lets through completely defined functions which nevertheless do not use all of their arguments. This is somewhat unfortunate. The problem could be circumvented (at the expense of complicating the program slightly) by ensuring that all constraints (symmetry-breaking or otherwise) are called when we reach a completely defined function. However, though this would get rid of the immediate problem, it would not help with the case where we output an incomplete function specification, none of whose induced functions are essentially n-ary. In other words, we could ameliorate the problem slightly by complicating the code, but the only way to completely remove it would be to remove the dependency on the `complete` flag.

The one remaining question is: how can we tell if an argument has been used essentially? This is quite easy. Whenever we restrict the value of some $f(\mathbf{t})$, we perform the following algorithm.

foreach argument i of f
  if $(\bigcap\{f(\mathbf{t}') : \mathbf{t}'$ differs from $\mathbf{t}$ in the i^{th} place $\} = \emptyset)$
    then argument i is used essentially.

Of course, once we have proved an argument to be essential at some level of the search tree, there is no need to test it at deeper levels. The **foreach** loop can skip such arguments.

There is one more heuristic which we could (but don't) make use of. If $f(\mathit{some\_tuple})$ can take a greater number of possible values than there are arguments which we have not yet decided are essential, then it must be able to take some value which will make essential use of all the arguments. The proof of this is left to the reader.

4.2.3 Constraints with Nested Function Applications

In this section, we turn our attention to constraints such as the `AssociativeConstraint` which have the characteristic that their definition as identities involves nested applications of function calls. In the case of associativity itself, there are two nested applications of f in the defintion

$$\forall x \forall y \forall z f(f(x,y),z) = f(x,f(y,z)).$$

How might we go about implementing such a constraint? It takes only a moment's thought to realise the difficulties. In general, $f(x,y)$ might take any one of several values; how do these values affect the possible values taken by $f(y,z)$? If the solving algorithm has determined that $f(1,3) \in \{1,0,2\}$, say, what can we say about the values of f when applied to other tuples?

These questions can be answered in general, but they give rise to algorithms which are relatively slow to execute, and tedious to program. The algorithms emplyed in Polyanna as it currently stands are rather simpler, and run faster (at least on domain sizes up to about 10). On the other hand, the more general algorithms may be faster on large domain sizes (though "faster" might still mean "prohibitively slowly").

The associative constraint in Polyanna does not do anything unless both $f(x, y)$ and $f(y, z)$ are singleton sets. In this simple case, the algorithm is trivial: the `AssociativeConstraint` simply ensures that $f(f(x, y), z)$ and $f(x, f(y, z))$ agree on the values that they can take.

Other constraints which involve nested function applications are the `Affine1Constraint` ($f(f(x, y, z), z, u) = f(x, y, u)$) and the `TwoSemilattice1Constraint` ($f(x, f(y, x)) = f(y, x)$). These are handled in the same way as the `AssociativeConstraint`.

References

[1] R. Gault and P. Jeavons. Implementing a test for tractability, 2001. Submitted for publication.

[2] P. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200:185–204, 1998.