Conjure Documentation, Release 2.4.0

Release 2.4.0

Özgür Akgün, Saad Attieh, Juliana Bowles, Nguyen Dang, Joan E

Nov 21, 2022

TABLE OF CONTENTS

1 Welcome **1**

2 Introduction **3**

3 Installation **5**
- 3.1 Downloading a binary . . . 5
- 3.2 Compiling from source . . . 5
- 3.3 Installing Savile Row . . . 6
- 3.4 Compiling the documentation . . . 6

4 Command Line Interface **7**
- 4.1 Help output . . . 8

5 Features **21**
- 5.1 Problem classes . . . 21
- 5.2 High level of abstraction . . . 21
- 5.3 Arbitrarily nested types . . . 22
- 5.4 Automatic symmetry breaking . . . 22
- 5.5 Multiple models . . . 22
- 5.6 Automated channelling . . . 23
- 5.7 Extensibility . . . 23
- 5.8 Multiple target solvers . . . 23

6 Conjure's input language: Essence **25**
- 6.1 Declarations . . . 26
  - 6.1.1 Declaring decision variables . . . 26
  - 6.1.2 Declaring parameters . . . 26
  - 6.1.3 Declaring aliases . . . 26
  - 6.1.4 Declaring enumerated types . . . 27
  - 6.1.5 Declaring unnamed types . . . 27
- 6.2 Branching statements . . . 27
- 6.3 Constraints . . . 28
- 6.4 Instantiation conditions . . . 28
- 6.5 Objective statements . . . 28
- 6.6 Names . . . 28
- 6.7 Domains . . . 28
  - 6.7.1 Boolean domains . . . 29
  - 6.7.2 Integer domains . . . 29
  - 6.7.3 Enumerated domains . . . 30
  - 6.7.4 Unnamed domains . . . 30
  - 6.7.5 Tuple domains . . . 30

6.7.6 Record domains . . . 30
6.7.7 Variant domains . . . 31
6.7.8 Matrix domains . . . 31
6.7.9 Set domains . . . 31
6.7.10 Multi-set domains . . . 32
6.7.11 Function domains . . . 32
6.7.12 Sequence domains . . . 32
6.7.13 Relation domains . . . 33
6.7.14 Partition domains . . . 33
6.8 Types . . . 34
6.9 Expressions . . . 34
6.9.1 Matrix indexing . . . 34
6.9.2 Tuple indexing . . . 35
6.9.3 Arithmetic operators . . . 35
6.9.4 Comparisons . . . 36
6.9.5 Logical operators . . . 37
6.9.6 Set operators . . . 37
6.9.7 Sequence operators . . . 38
6.9.8 Enumerated type operators . . . 38
6.9.9 Multiset operators . . . 38
6.9.10 Type conversion operators . . . 39
6.9.11 Function operators . . . 39
6.9.12 Matrix operators . . . 40
6.9.13 Partition operators . . . 40
6.9.14 List combining operators . . . 41
6.9.15 Comprehensions . . . 41
6.9.16 Miscellaneous examples . . . 42

7 Tutorials 43
7.1 Number puzzle . . . 43
7.1.1 Initial model . . . 43
7.1.2 Identifying a missing constraint . . . 44
7.1.3 Final model . . . 44
7.2 The Knapsack problem . . . 45
7.3 Nurse rostering . . . 47
7.3.1 Initial specification . . . 47
7.3.2 Changing an overly restrictive assumption . . . 48
7.3.3 Final model . . . 49
7.4 Labelled connected graphs . . . 49
7.4.1 Model 1: distance matrix . . . 50
7.4.2 Model 2: reachability matrix . . . 51
7.4.3 Model 3: structured reachability matrices . . . 52
7.4.4 Model 4: connected component . . . 53
7.4.5 Model 5: minimal connected component . . . 54
7.4.6 Generating all connected graphs . . . 54
7.5 Futoshiki . . . 55
7.5.1 Problem . . . 55
7.5.2 Essence Model . . . 56
7.5.3 Instance . . . 57
7.5.4 Solving . . . 58
7.6 BIBD . . . 58
7.6.1 The Problem . . . 58
7.6.2 The Model . . . 59
7.6.3 Improvements . . . 62

7.7 Semigroups, Monoids and Groups . . . 62
7.7.1 The Problem . . . 62
7.7.2 Semigroups . . . 63
7.7.3 Monoids . . . 64
7.7.4 Groups . . . 64
7.8 Handcrafting Instance Generators in Essence . . . 66
7.8.1 Instances for the Knapsack problem . . . 66
7.9 Simple Permutations . . . 69
7.9.1 Problem . . . 69
7.9.2 Enumeration/Generation Model . . . 69
7.9.3 Parameter . . . 70
7.9.4 Solving . . . 70
7.9.5 Checking Model . . . 70
7.9.6 Instances . . . 71
7.9.7 Solving . . . 71

8 Contact **73**
8.1 Contributors . . . 73

Bibliography **75**

CHAPTER

ONE

WELCOME

Welcome to the documentation of Conjure!

Conjure is an automated modelling tool for Constraint Programming.

In this documentation, you will find the following.

- A brief introduction to Conjure,
- installation instructions,
- a description of how to use Conjure through its command line user interface,
- a list of Conjure's features,
- a description of Conjure's input language Essence, and
- a collection of simple demonstrations of Conjure's use.

CHAPTER

TWO

INTRODUCTION

Conjure is an automated constraint modelling tool for Constraint Programming.

Its input language, Essence, is a high level problem specification language. Essence allows writing problem specifications at a high level of abstraction and without having to make a lot of low level modelling decisions.

Conjure reads in abstract problem specifications (in Essence) and produces concrete constraint programming models (in Essence'). Essence' is a solver independent constraint modelling language. Using the Savile Row tool, an Essence' model can be instantiated with parameter values and solved using one of several backends. More information on Savile Row can be found on its website.

Conjure works at the problem class level. A problem class is a parameterised specification of a problem; it does not encode a single problem but a class of problems. For example, a problem specification for the game of Sudoku is typically parameterised over the hints (the prefilled cells). A problem specification (or model) at the class level is said to be *instantiated* when values are provided for its parameters. In the case of a Sudoku, the parameter values are the contents of the hint cells.

Operating at the class level has one very important benefit: Conjure needs to be executed only once to create one (or more) Essence' models for a problem. Once the models are generated, they can be used to solve many instances of the same class.

CHAPTER

THREE

INSTALLATION

Conjure can be installed either by downloading a binary distribution, or by compiling it from source code.

3.1 Downloading a binary

Conjure is available as an executable binary for Linux and MacOS. If it is available for your platform, you can just download it and run it. It may be useful to save the binary under a directory that is in your search PATH, so you do not have to type the full path to the Conjure executable to run it.

For Windows, please use the Linux binaries with the Windows Subsystem for Linux.

3.2 Compiling from source

In order to compile Conjure on your computer, please download the source code from GitHub.

```
git clone git@github.com:conjure-cp/conjure.git
cd conjure
BIN_DIR=/somewhere/in/your/path make install
```

Conjure is implemented in Haskell, it can be compiled using either cabal-install or stack.

It comes with a Makefile which will use Stack by default. The default target in the Makefile will install Stack using the standard procedures (which involves downloading and running a script). For more precise control, you might want to consider installing the Haskell tools beforehand instead of using the Makefile.

Installation is known to work with GHC-8.0.2, GHC-8.2.2, GHC-8.4.4, and GHC-8.6.5.

In addition, a number of supported backend solvers can be compiled using the *make solvers* target. This target also takes a BIN_DIR environment variable to control the location of the solver executables, and a PROCESSES environment variable to control how many processes to use when building solvers

```
BIN_DIR=/somewhere/in/your/path PROCESSES=4 make solvers
```

3.3 Installing Savile Row

Since Conjure works by generating an Essence' model, Savile Row is a vital tool when using it. Savile Row can be downloaded from its website.

You do not need to download Savile Row separately when you compile Conjure from source. An up-to-date version of Savile Row is also copied next to the Conjure executable.

3.4 Compiling the documentation

The Makefile target *docs* builds the manual in PDF and HTML format. This requires some Python packages to already be installed. With a recent Python 3 the following should work to install the needed Python dependencies:

```
pip3 install sphinx-rtd-theme sphinxcontrib-bibtex
```

Without specifying versions this currently installs Sphinx-5.x and recent versions of related dependencies. However, `sphinx-rtd-theme` forces `opendoc < 0.18` rather than a more recent version. Some packages used by `sphinxcontrib-bibtex` also seem to require an older version `opendoc` but do not explicitly declare this dependency, and just running `pip3 install sphinxcontrib-bibtex` will install packages which cannot successfully build the PDF documentation.

The PDF target needs a fairly comprehensive TeX installation, with many styles and fonts beyond a minimal installation. The default packages installed by MacTeX should work on macOS.

CHAPTER

FOUR

COMMAND LINE INTERFACE

Conjure supports a number of commands. A command is provided as the first argument to Conjure on the command line. It is followed by a number of mandatory arguments (if any) depending on the command, and a number of optional arguments.

Some command line arguments to Conjure are positional, for example the command name. Another example of positional arguments is the path to a file required by the `conjure pretty` command. This argument can just be provided after the command name, like: `conjure pretty myfile.essence`.

Non-positional arguments are provided using options. Some options require an additional value to be provided. Other options are flags and do not expect additional values. For example the `conjure pretty` command takes a flag called `--remove-unused`, which removes unused decision variables from the model before pretty printing it. This option is a flag that takes no values. However, the `conjure modelling` command takes an option called `--output-directory`, which specifies the directory under which Conjure places its output files. This option requires a value.

Options can have short or long names. Following the common convention, short option names are preceded by a single dash and long options names are preceded by two dashes. For example `--output-directory` is a long name for an option, and `-o` is a short name for the same option.

The general form of a Conjure run is as follows: `conjure [COMMAND] ... [OPTIONS]`.

Following is the list of primary commands provided by Conjure. They can be used to generate Essence' models from Essence files, translate parameter files and solution files for a specific Essence' model, and more.

modelling
: The main act. Given a problem specification in Essence, produce constraint programming models in Essence'.

translate-parameter
: Refinement of parameter files written in Essence for a particular Essence' model. The Essence' model needs to be generated by Conjure.

translate-solution
: Translation of solutions back to Essence.

validate-solution
: Validating a solution.

solve
: This is a combined mode, and it is available for convenience. It runs conjure in the modelling mode followed by parameter refinement if required, then Savile Row + Minion to solve, and then solution translation.

If no primary command is provided, `modelling` is assumed.

Conjure also supports a few additional commands on top of the primary commands listed above. These commands are not required for the normal operation of the tool. They are implemented to aid development and testing.

pretty
: Pretty print as Essence file to stdout. This mode can be used to view a binary Essence file in textual form.

diff
Diff on two Essence files. Works on models, parameters, and solutions.

type-check
Type-checking a single Essence file.

split
Split an Essence files to various smaller files. Useful for testing.

symmetry-detection
Dump some JSON to be used as input to ferret for symmetry detection.

parameter-generator
Generate an Essence model describing the instances of the problem class defined in the input Essence model. An error will be printed if the model has infinitely many instances.

Commands typically take additional arguments. Each command provides a separate help message. To see the command specific help message, run: `conjure COMMAND --help`.

4.1 Help output

The following is Conjure's full help message for each command, provided for reference. These messages may change between releases of Conjure.

```
Conjure: The Automated Constraint Modelling Tool

conjure [COMMAND] ... [OPTIONS]
  The command line interface of Conjure takes a command name as the first argument␣
↪followed by more arguments depending
  on the command.
  This help text gives a list of the available commands.
  For details of a command, pass the --help flag after the command name.
  For example: 'conjure translate-solution --help'

Common flags:
     --help                                 Display help message
     --version                              Print version information

conjure [modelling] [OPTIONS] ESSENCE_FILE
  The main act. Given a problem specification in Essence, produce constraint␣
↪programming models in Essence'.

Logging & Output:
  -o --output-directory=DIR                 Where to save generated models.
                                            Default value: 'conjure-output'
     --numbering-start=INT                  Starting value for output files.
                                            Default value: 1
     --smart-filenames                      Use "smart names" for models.
                                            Directs Conjure to use the answers when␣
↪producing a filename and to ignore
                                            the order of questions. Only useful if 'f
↪' is used for questions.
     --log-level=LOGLEVEL                   Log level.
     --verbose-trail                        Generate verbose trails.
     --rewrites-trail                       Generate trails about the applied␣
↪rewritings.
     --log-rule-fails                       Generate logs for rule failures.␣
```

(continues on next page)

(continued from previous page)

```
↪(Caution: can be a lot!)
     --log-rule-successes                        Generate logs for rule applications.
     --log-rule-attempts                         Generate logs for rule attempts.␣
↪(Caution: can be a lot!)
     --log-choices                               Store the choices in a way that can be␣
↪reused by -al
     --output-format=FORMAT                      Format to use for output. All output␣
↪formats can also be used for input.
                                                     plain: default
                                                     binary: a binary encoding
                                                     astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                     json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                     jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                             solutions are saved in a single file as␣
↪json, this mode prints one
                                             solution per line
                                                     minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                            Line width for pretty printing.
                                             Default: 120
Model generation:
     --responses=ITEM                            A comma separated list of integers.
                                             If provided, these will be used as the␣
↪answers during interactive model
                                             generation instead of prompting the user.
     --responses-representation=ITEM             A comma separated list of variable name␣
↪: integer pairs.
                                             If provided, these will be used as the␣
↪answers during interactive model
                                             generation instead of prompting the user␣
↪for the variable representation
                                             questions.
                                             See --dump-representations for a list of␣
↪available representation options.
     --estimate-number-of-models                 Calculate (a lower bound on) the number␣
↪of models, instead of running
                                             the usual modelling mode.
     --portfolio=PORTFOLIO                       Portfolio size. When it is set to N,␣
↪Conjure will try to generate up to
                                             N models.
  -q --strategy-q=STRATEGY                       Strategy for selecting the next question␣
↪to answer. Options: f (for
                                             first), i (for interactive), r (for␣
↪random), x (for all). Prepend a (for
                                             auto) to automatically skip when there␣
↪is only one option at any point.
                                             Default value: f
  -a --strategy-a=STRATEGY                       Strategy for selecting an answer. Same␣
↪options as strategy-q.
                                              c picks the most 'compact' option at␣
↪every decision point.
                                              s picks the 'sparsest' option at every␣
↪decision point: useful for
                                             --representations-givens
```

(continues on next page)

(continued from previous page)

```
                                            Default value: ai
     --representations=STRATEGY             Strategy for choosing a representation.
                                            Default value: same as --strategy-a
     --representations-finds=STRATEGY       Strategy for choosing a representation␣
↪for a decision variable.
                                            Default value: same as --representations
     --representations-givens=STRATEGY      Strategy for choosing a representation␣
↪for a parameter.
                                            Default value: s (for sparse)
     --representations-auxiliaries=STRATEGY Strategy for choosing a representation␣
↪for an auxiliary variable.
                                            Default value: same as --representations
     --representations-quantifieds=STRATEGY Strategy for choosing a representation␣
↪for a quantified variable.
                                            Default value: same as --representations
     --representations-cuts=STRATEGY        Strategy for choosing a representation␣
↪for cuts in 'branching on'.
                                            Default value: same as --representations
     --channelling                          Whether to produce channelled models␣
↪(true by default).
     --representation-levels                Whether to use built-in precedence␣
↪levels when choosing representations.
                                            Used to cut down the number of generated␣
↪models.
                                            Default: true
     --follow-model=ITEM                    Provide a Conjure-generated Essence␣
↪Prime model to be used as a guide
                                            during model generation. Conjure will␣
↪try to imitate the modelling
                                            decisions from this file.
     --seed=INT                             Random number generator seed.
     --limit-models=INT                     Maximum number of models to generate.
     --choices=FILE                         Choices to use for -al, either an eprime␣
↪file (created by
                                            --log-choices), or a json file.
General:
     --limit-time=INT                       Limit in seconds of real time.

conjure translate-parameter [OPTIONS]
  Refinement of Essence parameter files for a particular Essence' model.
  The model needs to be generated by Conjure.

Flags:
     --eprime=ESSENCE_FILE                  An Essence' model generated by Conjure.
     --essence-param=FILE                   An Essence parameter for the original␣
↪problem specification.
     --eprime-param=FILE                    An Essence' parameter matching the␣
↪Essence' model.
                                            Default is 'foo.eprime-param' if the␣
↪Essence parameter file is named
                                            'foo.param'.
Logging & Output:
     --log-level=LOGLEVEL                   Log level.
     --output-format=FORMAT                 Format to use for output. All output␣
↪formats can also be used for input.
                                                plain: default
                                                binary: a binary encoding
```

(continues on next page)

(continued from previous page)

```
                                                 astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                 json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                 jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                             solutions are saved in a single file as␣
↪json, this mode prints one
                                             solution per line
                                                 minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                        Line width for pretty printing.
                                             Default: 120
General:
     --limit-time=INT                        Limit in seconds of real time.

conjure translate-solution [OPTIONS]
  Translation of solutions back to Essence.

Flags:
     --eprime=FILE                           An Essence' model generated by Conjure.
                                             Mandatory.
     --essence-param=FILE                    An Essence parameter for the original␣
↪problem specification.
                                             Mandatory.
     --eprime-solution=FILE                  An Essence' solution for the␣
↪corresponding Essence' model.
     --essence-solution=FILE                 An Essence solution for the original␣
↪problem specification.
                                             By default, its value is the value of --
↪eprime-solution with extensions
                                             replaced by '.solution'.
Logging & Output:
     --log-level=LOGLEVEL                    Log level.
     --output-format=FORMAT                  Format to use for output. All output␣
↪formats can also be used for input.
                                                 plain: default
                                                 binary: a binary encoding
                                                 astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                 json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                 jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                             solutions are saved in a single file as␣
↪json, this mode prints one
                                             solution per line
                                                 minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                        Line width for pretty printing.
                                             Default: 120
General:
     --limit-time=INT                        Limit in seconds of real time.

conjure validate-solution [OPTIONS]
  Validating a solution.
```

(continues on next page)

(continued from previous page)

```
Flags:
     --essence=ESSENCE_FILE                         Problem specification in Essence.
     --param=FILE                                   Essence parameter file.
     --solution=FILE                                Essence solution.
Logging & Output:
     --log-level=LOGLEVEL                           Log level.
     --output-format=FORMAT                         Format to use for output. All output␣
↪formats can also be used for input.
                                                        plain: default
                                                        binary: a binary encoding
                                                        astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                        json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                        jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                                    solutions are saved in a single file as␣
↪json, this mode prints one
                                                    solution per line
                                                        minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                               Line width for pretty printing.
                                                    Default: 120
General:
     --limit-time=INT                               Limit in seconds of real time.

conjure solve [OPTIONS] ESSENCE_FILE [PARAMETER_FILE(s)]
  A combined mode for convenience.
  Runs Conjure in modelling mode followed by parameter translation if required, then␣
↪Savile Row + Minion to solve, and
  then solution translation.

General:
     --validate-solutions                           Enable solution validation.
     --limit-time=INT                               Limit in seconds of real time.
     --graph-solver                                 Create input files for the Glasgow graph␣
↪solver.
     --cgroups                                      Setup and use cgroups when solving with␣
↪Savile Row.
     --number-of-solutions=ITEM                     Number of solutions to find; "all"␣
↪enumerates all solutions.
                                                    Default: 1
     --copy-solutions                               Whether to place a copy of solution(s)␣
↪next to the Essence file or not.
                                                    Default: on
Logging & Output:
  -o --output-directory=DIR                         Where to save generated models.
                                                    Default value: 'conjure-output'
     --numbering-start=INT                          Starting value for output files.
                                                    Default value: 1
     --smart-filenames                              Use "smart names" for models.
                                                    Directs Conjure to use the answers when␣
↪producing a filename and to ignore
                                                    the order of questions. Only useful if 'f
↪' is used for questions.
     --solutions-in-one-file                        Place all solutions in a single file␣
↪instead of generating a separate
```

(continues on next page)

(continued from previous page)

```
                                                file per solution.
                                                Off by default.
     --log-level=LOGLEVEL                       Log level.
     --verbose-trail                            Generate verbose trails.
     --rewrites-trail                           Generate trails about the applied␣
↪rewritings.
     --log-rule-fails                           Generate logs for rule failures.␣
↪(Caution: can be a lot!)
     --log-rule-successes                       Generate logs for rule applications.
     --log-rule-attempts                        Generate logs for rule attempts.␣
↪(Caution: can be a lot!)
     --log-choices                              Store the choices in a way that can be␣
↪reused by -al
     --output-format=FORMAT                     Format to use for output. All output␣
↪formats can also be used for input.
                                                    plain: default
                                                    binary: a binary encoding
                                                    astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                    json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                    jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                                solutions are saved in a single file as␣
↪json, this mode prints one
                                                solution per line
                                                    minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                           Line width for pretty printing.
                                                Default: 120
Model generation:
     --responses=ITEM                           A comma separated list of integers.
                                                If provided, these will be used as the␣
↪answers during interactive model
                                                generation instead of prompting the user.
     --responses-representation=ITEM            A comma separated list of variable name␣
↪: integer pairs.
                                                If provided, these will be used as the␣
↪answers during interactive model
                                                generation instead of prompting the user␣
↪for the variable representation
                                                questions.
                                                See --dump-representations for a list of␣
↪available representation options.
     --portfolio=PORTFOLIO                      Portfolio size. When it is set to N,␣
↪Conjure will try to generate up to
                                                N models.
  -q --strategy-q=STRATEGY                      Strategy for selecting the next question␣
↪to answer. Options: f (for
                                                first), i (for interactive), r (for␣
↪random), x (for all). Prepend a (for
                                                auto) to automatically skip when there␣
↪is only one option at any point.
                                                Default value: f
  -a --strategy-a=STRATEGY                      Strategy for selecting an answer. Same␣
↪options as strategy-q.
                                                 c picks the most 'compact' option at␣
```

(continues on next page)

(continued from previous page)

```
↪every decision point.
                                              s picks the 'sparsest' option at every␣
↪decision point: useful for
                                             --representations-givens
                                             Default value: c
    --representations=STRATEGY               Strategy for choosing a representation.
                                             Default value: same as --strategy-a
    --representations-finds=STRATEGY         Strategy for choosing a representation␣
↪for a decision variable.
                                             Default value: same as --representations
    --representations-givens=STRATEGY        Strategy for choosing a representation␣
↪for a parameter.
                                             Default value: s (for sparse)
    --representations-auxiliaries=STRATEGY   Strategy for choosing a representation␣
↪for an auxiliary variable.
                                             Default value: same as --representations
    --representations-quantifieds=STRATEGY   Strategy for choosing a representation␣
↪for a quantified variable.
                                             Default value: same as --representations
    --representations-cuts=STRATEGY          Strategy for choosing a representation␣
↪for cuts in 'branching on'.
                                             Default value: same as --representations
    --channelling                            Whether to produce channelled models␣
↪(true by default).
    --representation-levels                  Whether to use built-in precedence␣
↪levels when choosing representations.
                                             Used to cut down the number of generated␣
↪models.
                                             Default: true
    --follow-model=ITEM                      Provide a Conjure-generated Essence␣
↪Prime model to be used as a guide
                                             during model generation. Conjure will␣
↪try to imitate the modelling
                                             decisions from this file.
    --seed=INT                               Random number generator seed.
    --limit-models=INT                       Maximum number of models to generate.
    --use-existing-models=FILE               File names of Essence' models generated␣
↪beforehand.
                                             If given, Conjure skips the modelling␣
↪phase and uses the existing models
                                             for solving.
                                             The models should be inside the output␣
↪directory (See -o).
Options for other tools:
    --savilerow-options=ITEM                 Options passed to Savile Row.
    --solver-options=ITEM                    Options passed to the backend solver.
    --solver=ITEM                            Backend solver. Possible values:
                                              - minion (CP solver)
                                              - gecode (CP solver)
                                              - chuffed (CP solver)
                                              - glucose (SAT solver)
                                              - glucose-syrup (SAT solver)
                                              - lingeling/plingeling/treengeling (SAT␣
↪solver)
                                              - cadical (SAT solver)
                                              - kissat (SAT solver)
                                              - minisat (SAT solver)
```

(continues on next page)

(continued from previous page)

```
                                                 - bc_minisat_all (AllSAT solver, only␣
↪works with
                                                --number-of-solutions=all)
                                                 - nbc_minisat_all (AllSAT solver, only␣
↪works with
                                                --number-of-solutions=all)
                                                 - open-wbo (MaxSAT solver, only works␣
↪with optimisation problems)
                                                 - coin-or (MIP solver, implemented via␣
↪MiniZinc)
                                                 - cplex (MIP solver, implemented via␣
↪MiniZinc)
                                                 - boolector (SMT solver, supported␣
↪logics: bv)
                                                 - yices (SMT solver, supported logics:␣
↪bv, lia, idl)
                                                 - z3 (SMT solver, supported logics: bv,␣
↪lia, nia, idl)
                                                Default: minion
                                                Default logic for SMT solvers is␣
↪bitvector (bv).
                                                Append a dash and the name of a logic to␣
↪the solver name to choose a
                                                different logic. For example yices-idl.

conjure ide [OPTIONS] ESSENCE_FILE
  IDE support features for Conjure.
  Not intended for direct use.

Logging & Output:
     --log-level=LOGLEVEL                       Log level.
     --line-width=INT                           Line width for pretty printing.
                                                Default: 120
General:
     --limit-time=INT                           Limit in seconds of real time.
IDE Features:
     --dump-declarations                        Print information about top level␣
↪declarations.
     --dump-representations                     List the available representations for␣
↪decision variables and
                                                parameters.

conjure pretty [OPTIONS] ESSENCE_FILE
  Pretty print as Essence file to stdout.
  This mode can be used to view a binary Essence file in textual form.

Flags:
     --normalise-quantified                     Normalise the names of quantified␣
↪variables.
     --remove-unused                            Remove unused declarations.
Logging & Output:
     --log-level=LOGLEVEL                       Log level.
     --output-format=FORMAT                     Format to use for output. All output␣
↪formats can also be used for input.
                                                    plain: default
                                                    binary: a binary encoding
                                                    astjson: a JSON dump of the internal␣
```

(continues on next page)

(continued from previous page)

```
↪data structures, quite verbose
                                                json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                            solutions are saved in a single file as␣
↪json, this mode prints one
                                            solution per line
                                                minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                       Line width for pretty printing.
                                            Default: 120
General:
     --limit-time=INT                       Limit in seconds of real time.

conjure diff [OPTIONS] FILE FILE
  Diff on two Essence files. Works on models, parameters, and solutions.

Logging & Output:
     --log-level=LOGLEVEL                   Log level.
     --output-format=FORMAT                 Format to use for output. All output␣
↪formats can also be used for input.
                                                plain: default
                                                binary: a binary encoding
                                                astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                            solutions are saved in a single file as␣
↪json, this mode prints one
                                            solution per line
                                                minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                       Line width for pretty printing.
                                            Default: 120
General:
     --limit-time=INT                       Limit in seconds of real time.

conjure type-check [OPTIONS] ESSENCE_FILE
  Type-checking a single Essence file.

Logging & Output:
     --log-level=LOGLEVEL                   Log level.
General:
     --limit-time=INT                       Limit in seconds of real time.

conjure split [OPTIONS] ESSENCE_FILE
  Split an Essence file to various smaller files. Useful for testing.

Logging & Output:
  -o --output-directory=DIR                 Where to save generated models.
                                            Default value: 'conjure-output'
     --log-level=LOGLEVEL                   Log level.
     --output-format=FORMAT                 Format to use for output. All output␣
↪formats can also be used for input.
```

(continues on next page)

(continued from previous page)

```
                                             plain: default
                                             binary: a binary encoding
                                             astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                             json: a simplified JSON format, only␣
↪used for parameters and solutions
                                             jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                         solutions are saved in a single file as␣
↪json, this mode prints one
                                         solution per line
                                             minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                    Line width for pretty printing.
                                         Default: 120
General:
     --limit-time=INT                    Limit in seconds of real time.

conjure symmetry-detection [OPTIONS] ESSENCE_FILE
  Dump some JSON to be used as input to ferret for symmetry detection.

Logging & Output:
     --json=JSON_FILE                    Output JSON file.
                                         Default is 'foo.essence-json'
                                         if the Essence file is named 'foo.essence
↪'
     --log-level=LOGLEVEL                Log level.
     --output-format=FORMAT              Format to use for output. All output␣
↪formats can also be used for input.
                                             plain: default
                                             binary: a binary encoding
                                             astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                             json: a simplified JSON format, only␣
↪used for parameters and solutions
                                             jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                         solutions are saved in a single file as␣
↪json, this mode prints one
                                         solution per line
                                             minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                    Line width for pretty printing.
                                         Default: 120
General:
     --limit-time=INT                    Limit in seconds of real time.

conjure parameter-generator [OPTIONS] ESSENCE_FILE
  Generate an Essence model describing the instances of the problem class defined in␣
↪the input Essence model.
  An error will be printed if the model has infinitely many instances.

Integer bounds:
     --MININT=INT                        The minimum integer value for the␣
↪parameter values.
                                         Default: 0
     --MAXINT=INT                        The maximum integer value for the␣
```

(continues on next page)

(continued from previous page)

```
↪parameter values.
                                             Default: 100

Logging & Output:
     --log-level=LOGLEVEL                    Log level.
     --output-format=FORMAT                  Format to use for output. All output␣
↪formats can also be used for input.
                                                 plain: default
                                                 binary: a binary encoding
                                                 astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                 json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                 jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                             solutions are saved in a single file as␣
↪json, this mode prints one
                                             solution per line
                                                 minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                        Line width for pretty printing.
                                             Default: 120
General:
     --limit-time=INT                        Limit in seconds of real time.

conjure autoig [OPTIONS] ESSENCE_FILE OUTPUT_FILE
  Generate an Essence model describing the instances of the problem class defined in␣
↪the input Essence model.
  An error will be printed if the model has infinitely many instances.

Flags:
     --generator-to-irace                    Convert the givens in a hand written␣
↪generator model to irace syntax.
     --remove-aux                            Remove lettings whose name start with Aux
Logging & Output:
     --log-level=LOGLEVEL                    Log level.
     --output-format=FORMAT                  Format to use for output. All output␣
↪formats can also be used for input.
                                                 plain: default
                                                 binary: a binary encoding
                                                 astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                 json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                 jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                             solutions are saved in a single file as␣
↪json, this mode prints one
                                             solution per line
                                                 minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                        Line width for pretty printing.
                                             Default: 120
General:
     --limit-time=INT                        Limit in seconds of real time.

conjure boost [OPTIONS] ESSENCE_FILE
  Strengthen an Essence model as described in "Reformulating Essence Specifications␣
```

(continues on next page)

(continued from previous page)

```
↪for Robustness",
  which aims to make search faster.

Logging & Output:
     --log-level=LOGLEVEL                        Log level.
     --log-rule-successes                        Generate logs for rule applications.
     --output-format=FORMAT                      Format to use for output. All output␣
↪formats can also be used for input.
                                                     plain: default
                                                     binary: a binary encoding
                                                     astjson: a JSON dump of the internal␣
↪data structures, quite verbose
                                                     json: a simplified JSON format, only␣
↪used for parameters and solutions
                                                     jsonstream: same as JSON, except in␣
↪one special case. when multiple
                                                 solutions are saved in a single file as␣
↪json, this mode prints one
                                                 solution per line
                                                     minizinc: minizinc format for data␣
↪files, only used for solutions
     --line-width=INT                            Line width to use during pretty printing.
                                                 Default: 120
General:
     --limit-time=INT                            Time limit in seconds (real time).

conjure tsdef [OPTIONS]
  Generate data type definitions in TypeScript.
  These can be used when interfacing with Conjure via JSON.

Logging & Output:
     --log-level=LOGLEVEL                        Log level.
General:
     --limit-time=INT                            Limit in seconds of real time.
```

CHAPTER

FIVE

FEATURES

This section lists some features of Conjure.

Some of these are due to features of Conjure's input language Essence, and the need to support those. If you are not familiar with Essence, please see *Conjure's input language: Essence*.

5.1 Problem classes

Often, when we think of problems we think of a *class* of problems rather than a single problem. For example, Sudoku is a class of puzzles. There are many different Sudoku *instances*, with different clues. However, all Sudoku instances share the same set of rules. Describing the puzzle of Sudoku to somebody who doesn't know the rules of the game generally does not depend on a given set of clues.

Similarly, problem specifications in Essence are written for a class of problems instead of a single problem instance. For Sudoku, the rules (everything on a row/column/sub-grid has to be distinct) are encoded once. The clues are specified as *parameters* to the problem specification, together with appropriate assignment statements to incorporate the clues into the problem.

Many tools (solvers and/or modelling assistants) support this separation by having a parameterised problem specification in a file and separate data/parameter file specifying an instance of the problem. Conjure uses `*.essence` files for the problem specification, and `*.param` files for the parameter file.

Although a lot of tools support this kind of a separation, they generally work by instantiating a problem specification before operating on it. Conjure is different than most tools in this regard: it operates on parameterised problem specifications directly. It reads in a parameterised Essence file, and outputs one or more parameterised Essence' files. To solve an Essence' model provided by Conjure, Essence-level parameter files need to be translated to Essence'-level parameter files by running Conjure once per parameter file.

Savile Row accepts a parameterised model and a separate parameter file, and performs the instantiation. The output model and the translated parameter file from Conjure can be directly used when running Savile Row.

5.2 High level of abstraction

Conjure's input language is Essence. Essence provides abstract domain types like sets, multi-sets, functions, sequences, relations, partitions, records, and variants. These abstract domain types also support domain attributes like cardinality for set-like domains and injectivity/surjectivity for functions, to enable concise specification of a problem. Essence also provides more primitive domain types like Booleans, integers, enumerated types, and matrices, that are supported by most CP solvers and modelling assistants.

In addition to abstract domain types, Essence also provides operators that operate on parameters or decision variables with abstract domains. For example, set membership, subset, function inverse, and relation projection are provided to enable specification of problem constraints abstractly.

The high level of abstraction offered by Essence allows its users to specify problems without having to make a lot of low level *modelling decisions*.

5.3 Arbitrarily nested types

The abstract domain types provided by Essence are domain constructors: they take another domain as an argument to construct a new domain. For example the domain `set of D` represents a set of values from the domain `D`, and a `relation of (D1 * D2 * D3)` represents a relation between values of domains `D1`, `D2`, and `D3`.

Using these domain constructors, domains of arbitrary nesting can be created. Conjure does not have a limit on the level of nesting in the domains. But keep in mind: a several levels nested domain might look tiny whereas the combinatorial object it represents may be huge.

5.4 Automatic symmetry breaking

During its modelling process, a decision variable with an abstract domain type is *represented* using a collection of decision variables with more primitive domain types. For example the domain `set (size n) of D`, which represents a set of `n` values from the domain `D`, can be represented using the domain `matrix indexed by [int(1..n)] of D`. Performing this modelling transformation requires rewriting the rest of the model. Moreover, it introduces symmetry into the model, since a set implies a collection of distinct values whereas the matrix does not. To break this symmetry, Conjure introduces strict ordering constraints on adjacent entries of the matrix.

This is one of the simplest examples of automated symmetry breaking performed by Conjure. Conjure breaks all the symmetry introduced by modelling transformations like this one.

Another example is the domain `set of D` without the explicit `size` attribute. Since the number of elements in this set is not known, Conjure cannot simply use a matrix to represent this domain. There are multiple ways to represent this domain. One representation is to use an integer to partition the entries of the matrix into two: entries before the index pointed by this integer are regarded to be in the set, and entries after this position are regarded to be irrelevant.

It is important to post constraints on the irrelevant entries to fix them to a certain value. Not doing this introduces more symmetry. Conjure breaks this kind of symmetry by introducing constraints to fix their values.

5.5 Multiple models

Conjure is able to generate multiple Essence' models starting from a single Essence problem specification. Each model generated by Conjure can be used to solve the initial problem specified in Essence.

This feature is important because often a problem can be modelled in several ways, and it is difficult to know what a *good* model is for a given problem. Constraint programming experts spend considerable amounts of time developing models. It is common to create multiple models to compare how well they perform for a problem. A good model is then chosen only after several models have been considered.

Moreover, a single good model may not even exist for certain classes of problems. The choice of the model may depend on the instances we are interested in solving.

Lastly, instead of trying to pick a single good model a portfolio of models may be chosen with complementary strengths to exploit parallelism.

Conjure is able to produce multiple models mainly

- by having choices between multiple representations of decision variable domains, and
- by having choices between translating constraint expressions in multiple ways.

Both domain representation and constraint translation mechanisms are implemented using a rule based system inside Conjure to ease the addition of new modelling idioms.

5.6 Automated channelling

While modelling a problem using constraint programming, it is often possible to model a certain decision using multiple encodings. When different encodings with complementary strengths are available, experts can utilise this flexibility by using one encoding for parts of the formulation and another encoding for the rest of the formulation. When multiple encodings of a single decision are used in a single model, *channelling* constraints are added to ensure consistency between encodings.

In Conjure, decision variables with abstract domain types can very often be represented in multiple ways. For each occurrence of a decision variable, Conjure considers all representation options. If a decision variable is used more than once, this means that the decision variable can be represented in multiple ways in a single Essence' model.

When multiple representations are used, channelling constraints are generated by Conjure automatically. These constraints make sure that different representations of the same abstract combinatorial object have the same abstract value.

5.7 Extensibility

The modelling transformations of Conjure are implemented using a rule-based system.

There are two main kinds of rules in Conjure:

representations selection rules
to specify domain transformations,

expression refinement rules
to rewrite constraint expressions depending on their domain representations.

Moreover, Conjure contains a collection of **horizontal rules**, which are representation independent expression refinement rules. Thanks to horizontal rules, the number of representation dependent expression refinement rules are kept to a small number.

Conjure's architecture is designed to make adding both representation selection rules and expression refinement rules easy.

5.8 Multiple target solvers

The ability to target multiple solvers is not a feature of Conjure by itself, but a benefit it gains thanks to being a part of a state-of-the-art constraint programming tool-chain. Each Essence' model generated by Conjure can be solved using Savile Row together with one of its target solvers.

Savile Row can directly target Minion, Gecode (via fzn-gecode), and any SAT solver that supports the DIMACS format. It can also output Minizinc, and this output can be used to target a number of different solvers using the mzn2fzn tool.

CHAPTER

SIX

CONJURE'S INPUT LANGUAGE: ESSENCE

Conjure works on problem specifications written in Essence.

This section gives a description of Essence. A more thorough description can be found in the reference paper on Essence [FHJ+08].

We adopt a BNF-style format to describe all the constructs of the language. In the BNF format, we use the "#" character to denote comments, we use double-quotes for terminal strings, and we use a `list` construct to indicate a list of syntax elements.

The `list` construct has three arguments:

1. First argument is the syntax of the items of the list.
2. Second argument is the item separator.
3. Third argument indicates the surrounding bracket for the list. It can be one of round brackets (`()`), curly brackets (`{}`), or square brackets (`[]`).

Only the first argument is mandatory, the rest of the arguments are optional when an item separator or surrounding brackets are not required.

```
ProblemSpecification := list(Statement)
```

A problem specification in Essence is composed of a list of statements. Statements can declare decision variables, parameters or aliases. They can also post constraints, conditions on parameter values and an objective statement.

The order of statements is largely insignificant, except in one case: names need to be declared before use. For example a decision variable cannot be used before its declaration.

There are five kinds of statements in Essence.

```
Statement := DeclarationStatement
           | BranchingStatement
           | SuchThatStatement
           | WhereStatement
           | ObjectiveStatement
```

Every symbol must be declared before it is used, but otherwise statements can be listed in any order. A problem specification can contain at most one branching statement. A problem specification can contain at most one objective statement.

6.1 Declarations

```
DeclarationStatement := FindStatement
                      | GivenStatement
                      | LettingStatement
                      | GivenEnum
                      | LettingEnum
                      | LettingUnnamed
```

A declaration statement can be used to declare a decision variable (**FindStatement**), a parameter (**GivenStatement**), an alias for an expression or a domain (**LettingStatement**), and enumerated or unnamed types.

6.1.1 Declaring decision variables

```
FindStatement := "find" Name ":" Domain
```

A decision variable is declared by using the keyword `find`, followed by an identifier designating the name of the decision variable, followed by a colon symbol and the domain of the decision variable. The domains of decision variables have to be finite.

This detail is omitted in the BNF above for simplicity, but a comma separated list of names may also be used to declare multiple decision variables with the same domain in a single find statement. This applies to all declaration statements.

6.1.2 Declaring parameters

```
GivenStatement := "given" Name ":" Domain
```

A parameter is declared in a similar way to decision variables. The only difference is the use of the keyword `given` instead of the keyword `find`. Unlike decision variables, the domains of parameters do not have to be finite.

6.1.3 Declaring aliases

```
LettingStatement := "letting" Name "be" Expression
                  | "letting" Name "be" "domain" Domain
```

An alias for an expression can be declared by using the keyword `letting`, followed by the name of the alias, followed by the keyword `be`, followed by an expression. Similarly, an alias for a domain can be declared by including the keyword `domain` before writing the domain.

```
letting x be y + z
letting d be domain set of int(a..b)
```

In the example above `x` is declared as an expression alias for `y + z` and `d` is declared as a domain alias for `set of int(a..b)`.

6.1.4 Declaring enumerated types

```
GivenEnum := "given" Name "new type enum"

LettingEnum := "letting" Name "be" "new type enum" list(Name, ",", "{}")
```

Enumerated types can be declared in two ways: using a given-enum syntax or using a letting-enum syntax.

The given-enum syntax defers the specification of actual values of the enumerated type until instantiation. With this syntax, an enumerated type can be declared by only giving its name in the problem specification file. In a parameter file, values for the actual members of this type can be given. This allows Conjure to produce a model independent of the values of the enumerated type and only substitute the actual values during parameter instantiation.

The letting-enum syntax can be used to declare an enumerated type directly in a problem specification as well.

Values of an enumerated type cannot contain spaces.

```
letting direction be new type enum {North, East, South, West}
find x,y : direction
such that x != y
```

In the example fragment above `direction` is declared as an enumerated type with 4 members. Two decision variables are declared using `direction` as their domain and a constraint is posted on the values they can take. Enumerated types support equality, ordering, and successor/predecessor operators; they do not support arithmetic operators.

When an enumerated type is declared, the elements of the type are listed in increasing order.

6.1.5 Declaring unnamed types

```
LettingUnnamed := "letting" Name "be" "new type of size" Expression
```

Unnamed types are a feature of Essence which allow succinct specification of certain types of symmetry. An unnamed type is declared by giving it a name and a size (i.e. the number of elements in the type). The members of an unnamed type cannot be referred to individually. Typically constraints are posted using quantified variables over the whole domain. Unnamed types only support equality operators; they do not support ordering or arithmetic operators.

6.2 Branching statements

```
BranchingStatement := "branching" "on" list(BranchingOn, ",", "[]")

BranchingOn := Name
             | Expression
```

High level problem specification languages typically do not include lower level details such as directives specifying search order. Essence is such a language, and the reference paper on Essence ([FHJ+08]) does not include these search directives at all.

For pragmatic reasons Conjure supports search directives in the form of a branching-on statement, which takes a list of either variable names or expressions. Decision variables in a branching-on statement are searched using a static value ordering. Expressions can be used to introduce *cuts*; in which case when solving the model produced by Conjure, the solver is instructed to search for solutions satisfying the cut constraints first, and proceed to searching the rest of the search space later.

A problem specification can contain at most one branching statement.

6.3 Constraints

```
SuchThatStatement := "such that" list(Expression, ",")
```

Constraints are declared using the keyword sequence `such that`, followed by a comma separated list of Boolean expressions. The syntax for expressions is explained in section *Expressions*.

6.4 Instantiation conditions

```
WhereStatement := "where" list(Expression, ",")
```

Where statements are syntactically similar to *constraints*, however they cannot refer to decision variables. They can be used to post conditions on the parameters of the problem specification. These conditions are checked during parameter instantiation.

6.5 Objective statements

```
ObjectiveStatement := "minimising" Expression
                    | "maximising" Expression
```

An objective can be declared by using either the keyword `minimising` or the keyword `maximising` followed by an integer expression. A problem specification can have at most one objective statement. If it has none it defines a satisfaction problem, if it has one it defines an optimisation problem.

A problem specification can contain at most one objective statement.

6.6 Names

The lexical rules for valid names in Essence are similar to those of most common languages. A name consists of a sequence of non-whitespace alphanumeric characters (letters or digits) or underscores (_). The first character of a valid name has to be a letter or an underscore. Names are case-sensitive: Essence treats uppercase and lowercase versions of letters as distinct.

6.7 Domains

```
Domain := "bool"
        | "int" list(Range, ",", "()")
        | "int" "(" Expression ")"
        | Name list(Range, ",", "()") # the Name refers to an enumerated type
        | Name                         # the Name refers to an unnamed type
        | "tuple" list(Domain, ",", "()")
        | "record" list(NameDomain, ",", "{}")
        | "variant" list(NameDomain, ",", "{}")
        | "matrix indexed by" list(Domain, ",", "[]") "of" Domain
        | "set" list(Attribute, ",", "()") "of" Domain
        | "mset" list(Attribute, ",", "()") "of" Domain
```

(continues on next page)

(continued from previous page)

```
        | "function" list(Attribute, ",", "()") Domain "-->" Domain
        | "sequence" list(Attribute, ",", "()") "of" Domain
        | "relation" list(Attribute, ",", "()") "of" list(Domain, "*", "()")
        | "partition" list(Attribute, ",", "()") "from" Domain

Range := Expression
       | Expression ".."
       | ".." Expression
       | Expression ".." Expression

Attribute := Name
           | Name Expression

NameDomain := Name ":" Domain
```

Essence contains a rich selection of domain constructors, which can be used in an arbitrarily nested fashion to create domains for problem parameters, decision variables, quantified expressions and comprehensions. Quantified expressions and comprehensions are explained under *Expressions*.

Domains can be finite or infinite, but infinite domains can only be used when declaring of problem parameters. The domains for both decision variables and quantified variables have to be finite.

Some kinds of domains can take an optional list of attributes. An attribute is either a label or a label with an associated value. Different kinds of domains take different attributes.

Multiple attributes can be used in a single domain. Using contradicting values for the attribute values may result in an empty domain.

In the following, each kind of domain is described in a subsection of its own.

6.7.1 Boolean domains

The Boolean domain is denoted with the keyword `bool` and has two values: `false` and `true`. The Boolean domain is ordered with `false` preceding `true`. It is not currently possible to specify an objective with respect to a Boolean value. If `a` is a Boolean variable to minimise or maximise in the objective, use `toInt(a)` instead (see *Type conversion operators*).

6.7.2 Integer domains

An integer domain is denoted by the keyword `int`, followed by a list of integer ranges inside round brackets. The list of ranges is optional, if omitted the integer domain denotes the infinite domain of all integers.

An integer range is either a single integer, or a list of sequential integers with a given lower and upper bound. The bounds can be omitted to create an open range, but note that using open ranges inside an integer domain declaration creates an infinite domain.

Integer domains can also be constructed using a single set expression inside the round brackets, instead of a list of ranges. The integer domain contains all members of the set in this case. Note that the set expression cannot contain references to decision variables if this syntax is used.

Values in an integer domain should be in the range -2**62+1 to 2**62-1 as values outside this range may trigger errors in Savile Row or Minion, and lead to Conjure unexpectedly but silently deducing unsatisfiability. Intermediate values in an integer expression must also be inside this range.

6.7.3 Enumerated domains

Enumerated types are declared using the syntax given in *Declaring enumerated types*.

An enumerated domain is denoted by using the name of the enumerated type, followed by a list of ranges inside round brackets. The list of ranges is optional, if omitted the enumerated domain denotes the finite domain containing all values of the enumerated type.

A range is either a single value (member of the enumerated type), or a list of sequential values with a given lower and upper bound. The bounds can be omitted to create an open range, when an open range is used the omitted bound is considered to be the same as the corresponding bound of the enumerated type.

6.7.4 Unnamed domains

Unnamed types are declared using the syntax given in *Declaring unnamed types*.

An unnamed domain is denoted by using the name of the unnamed type. It does not take a list of ranges to limit the values in the domain, an unnamed domain always contains all values in the corresponding unnamed type.

6.7.5 Tuple domains

Tuple is a domain constructor, it takes a list of domains as arguments. Tuples can be of arbitrary arity.

A tuple domain is denoted by the keyword `tuple`, followed by a list of domains separated by commas inside round brackets. The keyword `tuple` is optional for tuples of arity greater or equal to 2.

When needed, domains inside a tuple are referred to using their positions. In an n-arity tuple, the position of the first domain is 1, and the position of the last domain is n.

To explicitly specify a tuple, use a list of values inside round brackets, preceded by the keyword `tuple`.

```
letting s be tuple()
letting t be tuple(0,1,1,1)
```

6.7.6 Record domains

Record is a domain constructor, it takes a list of name-domain pairs as arguments. Records can be of arbitrary arity. (A name-domain pair is a name, followed by a colon, followed by a domain.)

A record domain is denoted by the keyword `record`, followed by a list of name-domain pairs separated by commas inside curly brackets.

Records are very similar to tuples; except they use labels for their components instead of positions. When needed, domains inside a record are referred to using their labels.

To explicitly specify a record, use a list of values inside round brackets, preceded by the keyword `tuple`.

```
letting s be record{}
letting t be record{A : int(0..1), B : int(0..2)}
```

6.7.7 Variant domains

Variant is a domain constructor, it takes a list of name-domain pairs as arguments. Variants can be of arbitrary arity.

A variant domain is denoted by the keyword `variant`, followed by a list of name-domain pairs separated by commas inside curly brackets.

Variants are similar to records but with a very important distinction. A member of a record domain contains a value for each component of the record, however a member of a variant domain contains a value for only one of the components of the variant.

Variant domains are similar to tagged unions in other programming languages.

6.7.8 Matrix domains

Matrix is a domain constructor, it takes a list of domains for its indices and a domain for the entries of the matrix. Matrices can be of arbitrary dimensionality (greater than 0).

A matrix domain is denoted by the keywords `matrix indexed by`, followed by a list of domains separated by commas inside square brackets, followed by the keyword `of`, and another domain.

A matrix can be indexed only by integer, Boolean, or enumerated domains.

Matrix domains are the most basic container-like domains in Essence. They are used when the decision variable or the problem parameter does not have any further relevant structure. Using another kind of domain is more appropriate for most problem specifications in Essence.

Matrix domains are not ordered, but matrices can be compared using the equality operators. Note that two matrices are only equal if their indices are the same.

To explicitly specify a matrix, use a list of values inside square brackets. Optionally, the domain used to index the elements can be specified also.

```
letting M be [0,1,0,-1]
letting N be [[0,1],[0,-1]]
```

The matrix `[0,1]` is the same as `[0,1; int(1..2)]`, but distinct from `[0,1; int(0..1)]`.

6.7.9 Set domains

Set is a domain constructor, it takes a domain as argument denoting the domain of the members of the set.

A set domain is denoted by the keyword `set`, followed by an optional comma separated list of set attributes, followed by the keyword `of`, and the domain for members of the set.

Set attributes are all related to cardinality: `size`, `minSize`, and `maxSize`.

To explicitly specify a set, use a list of values inside curly brackets. Values only appear once in the set; if repeated values are specified then they are ignored.

```
letting S be {1,0,1}
```

6.7.10 Multi-set domains

Multi-set is a domain constructor, it takes a domain as argument denoting the domain of the members of the multi-set.

A multi-set domain is denoted by the keyword `mset`, followed by an optional comma separated list of multi-set attributes, followed by the keyword `of`, and the domain for members of the multi-set.

There are two groups of multi-set attributes:

1. Related to cardinality: `size`, `minSize`, and `maxSize`.
2. Related to number of occurrences of values in the multi-set: `minOccur`, and `maxOccur`.

Since a multi-set domain is infinite without a `size`, `maxSize`, or `maxOccur` attribute, one of these attributes is mandatory to define a finite domain.

To explicitly specify a multi-set, use a list of values inside round brackets, preceded by the keyword `mset`. Values may appear multiple times in a multi-set.

```
letting S be mset(0,1,1,1)
```

6.7.11 Function domains

Function is a domain constructor, it takes two domains as arguments denoting the *defined* and the *range* sets of the function. It is important to take note that we are using *defined* to mean the domain of the function, and *range* to mean the codomain.

A function domain is denoted by the keyword `function`, followed by an optional comma separated list of function attributes, followed by the two domains separated by an arrow symbol: `-->`.

There are three groups of function attributes:

1. Related to cardinality: `size`, `minSize`, and `maxSize`.
2. Related to function properties: `injective`, `surjective`, and `bijective`.
3. Related to partiality: `total`.

Cardinality attributes take arguments, but the rest of the arguments do not. Function domains are partial by default, and using the `total` attribute makes them total.

To explicitly specify a function, use a list of assignments, each of the form `input --> value`, inside round brackets and preceded by the keyword `function`.

```
letting f be function(0-->1,1-->0)
```

6.7.12 Sequence domains

Sequence is a domain constructor, it takes a domain as argument denoting the domain of the members of the sequence.

A sequence is denoted by the keyword `sequence`, followed by an optional comma separated list of sequence attributes, followed by the keyword `of`, and the domain for members of the sequence.

There are 2 groups of sequence attributes:

1. Related to cardinality: `size`, `minSize`, and `maxSize`.
2. Related to function-like properties: `injective`, `surjective`, and `bijective`.

Cardinality attributes take arguments, but the rest of the arguments do not. Sequence domains are total by default, hence they do not take a separate `total` attribute.

Sequences are indexed by a contiguous list of increasing integers, beginning at 1. The first value in a sequence `s` is `s(1)`.

To explicitly specify a sequence, use a list of values inside round brackets, preceded by the keyword `sequence`.

```
letting s be sequence(1,0,-1,2)
letting t be sequence() $ empty sequence
```

6.7.13 Relation domains

Relation is a domain constructor, it takes a list of domains as arguments. Relations can be of arbitrary arity.

A relation domain is denoted by the keyword `relation`, followed by an optional comma separated list of relation attributes, followed by the keyword `of`, and a list of domains separated by the `*` symbol inside round brackets.

There are 2 groups of relation attributes:

1. Related to cardinality: `size`, `minSize`, and `maxSize`.
2. Binary relation attributes: `reflexive`, `irreflexive`, `coreflexive`, `symmetric`, `antiSymmetric`, `aSymmetric`, `transitive`, `total`, `connex`, `Euclidean`, `serial`, `equivalence`, `partialOrder`.

The binary relation attributes are only applicable to relations of arity 2, and are between two identical domains.

To explicitly specify a relation, use a list of tuples, enclosed by round brackets and preceded by the keyword `relation`. All the tuples must be of the same type.

```
letting R be relation((1,1,0),(1,0,1),(0,1,1))
```

6.7.14 Partition domains

Partition is a domain constructor, it takes a domain as an argument denoting the members in the partition.

A partition is denoted by the keyword `partition`, followed by an optional comma separated list of partition attributes, followed by the keyword `from`, and the domain for the members in the partition.

There are 3 groups of partition attributes:

1. Related to the number of parts in the partition: `numParts`, `minNumParts`, and `maxNumParts`.
2. Related to the cardinality of each part in the partition: `partSize`, `minPartSize`, and `maxPartSize`.
3. Partition properties: `regular`.

The first and second groups of attributes are related to number of parts and cardinalities of each part in the partition. The `regular` attribute forces each part to be of the same cardinality without specifying the actual number of parts or cardinalities of each part.

6.8 Types

Essence is a statically typed language. A declaration – whether it is a decision variable, a problem parameter or a quantified variable – has an associated domain. From its domain, a type can be calculated.

A type is obtained from a domain by removing attributes (from set, multi-set, function, sequence, relation, and partition domains), and removing bounds (from integer and enumerated domains).

In the expression language of Essence, each operator has a typing rule associated with it. These typing rules are used to both type check expression fragments and to calculate the types of resulting expressions.

For example, the arithmetic operator + requires two arguments both of which are integers, and the resulting expression is also an integer. So if `a`, and `b` are integers `a + b` is also an integer. Conjure gives a type error otherwise.

Using these typing rules every Essence expression can be checked for type correctness statically.

6.9 Expressions

```
Expression := Literal
            | Name
            | Quantification
            | Comprehension Expression [GeneratorOrCondition]
            | Operator

Operator := ...
```

(In preparation)

6.9.1 Matrix indexing

A list is a one-dimensional matrix indexed by an integer, starting at 1. Matrices of dimension k are implemented by a list of matrices of dimension k-1.

```
letting D1 be domain matrix indexed by [int(1..2),int(1..5)] of int(-1..1)
letting E be domain matrix indexed by [int(1..5)] of int(-1..1)
letting D2 be domain matrix indexed by [int(1..2)] of E
find A : D1 such that A[1] = [-1,1,1,0,1], A[2] = [1,1,1,1,1]
find B : D2 such that B[1] = A[1], B[2] = [0,0,0,0,0]
letting C be [[-1,1,1,0,1],[0,0,0,0,0]]
letting a be A[1][1] = -1                   $ true
letting b be A[1,1] = -1                    $ true
letting c be C[1] = [-1,1,1,0,1]            $ true
letting d be B[1] = C[1]                    $ true
letting e be [A[1],B[2]] = C                $ true
letting f be B = C                          $ true
letting F be domain matrix indexed by [int(1..6)] of bool
find g : F such that g = [a,b,c,d,e,f] $ [true,true,true,true,true,true]
```

6.9.2 Tuple indexing

Tuples are indexed by a constant integer, starting at 1. The first value in a tuple `t` is `t[1]`. Attempting to access a tuple element via an index that is negative, zero, or too large for the tuple, results in an error.

```
letting s be tuple(0,1,1,0)
letting t be tuple(0,0,0,1)
find a : bool such that a = (s[1] = t[1]) $ true
```

6.9.3 Arithmetic operators

Essence supports the four usual arithmetic operators

    + - * /

and also the modulo operator `%`, exponentiation `**`. These all take two arguments and are expressed in infix notation.

There is also the unary prefix operator `-` for negation, the unary postfix operator `!` for the factorial function, and the absolute value operator `|x|`.

The arithmetic operators have the usual precedence: the factorial operator is applied first, then exponentiation, then negation, then the multiplication, division, and modulo operators, and finally addition and subtraction.

Exponentiation associates to the right, other binary operators to the left.

Division

Division returns an integer, and the following relationship holds when `x` and `y` are integers and `y` is not zero:

    (x % y) + y*(x / y) = x

whenever `y` is not zero. `x / 0` and `x % 0` are expressions that do not have a defined value. Division by zero may lead to unsatisfiability but is not flagged by either Conjure or Savile Row as an error.

Factorial

Both `factorial(x)` and `x!` denote the product of all positive integers up to `x`, with `x! = 1` whenever `x <= 0`. The factorial operator cannot be used directly in expressions involving decision variables, so the following

```
find z : int(-1..13)
such that (z! > 2**28)
```

is flagged as an error. However, the following does work:

```
find z : int(-1..13)
such that (exists x : int(-1..13) . (x! > 2**28) /\ (z=x))
```

Powers

When `x` is an integer and `y` is a positive integer, then `x**y` denotes `x` raised to the `y`-th power. When `y` is a negative integer, `x**y` is flagged by Savile Row as an error (this includes `1**(-1)`). Conjure does not flag negative powers as errors. The relationship

```
x ** y = x*(x**(y-1))
```

holds for all integers `x` and positive integers `y`. This means that `x**0` is always 1, whatever the value of `x`.

Negation

The unary operator `-` denotes negation; when `x` is an integer then `--x = x` is always true.

Absolute value

When `x` is an integer, `|x|` denotes the absolute value of `x`. The relationship

```
(2*toInt(x >= 0) - 1)*x = |x|
```

holds for all integers `x` such that `|x| <= 2**62-2`. Integers outside this range may be flagged as an error by Savile Row and/or Minion.

6.9.4 Comparisons

The inline binary comparison operators `=` `!=` `<` `<=` `>` `<=` can be used to compare two expressions.

The equality operators `=` and `!=` can be applied to compare two expressions, both taking values in the same domain. Equality operators are supported for all types.

The equality operators have the same precedence as other logical operators. This may lead to unintended unsatisfiability or introducing inadvertent solutions. This is illustrated in the following example, where there are two possible solutions.

```
find a : bool such that a = false  \/ true  $ true or false
find b : bool such that b = (false \/ true) $ true
```

The inline binary comparison operators `<` `<=` `>` `<=` can be used to compare expressions taking values in an ordered domain. The expressions must both be integer, both Boolean or both enumerated types.

```
letting direction be new type enum {North, East, South, West}
find a : bool such that a = ((North < South) /\ (South < West))  $ true
find b : bool such that b = (false <= true) $ true
```

The inline binary comparison operators

```
<lex <=lex >lex >=lex
```

test whether their arguments have the specified relative lexicographic order.

```
find v : matrix indexed by [int(1..2)] of int(1..2)
such that v <lex [ v[3-i] | i : int(1..2) ] $ v = [1,2]
```

6.9.5 Logical operators

/\	and
\/	or
->	implication
<->	if and only if
!	negation

Logical operators operate on Boolean valued expressions, returning a Boolean value `false` or `true`. Negation is unary prefix, the others are binary inline. The `and`, `or` and `xor` operators can be applied to sets or lists of Boolean values (see *List combining operators* for details). Note that `<-` is not a logical operator, but is used in list comprehension syntax.

6.9.6 Set operators

The following set operators return Boolean values indicating whether a specific relationship holds:

`in`	test if element is in set
`subset`	test if first set is strictly contained in second set
`subsetEq`	test if first set is contained in second set
`supset`	test if first set strictly contains second set
`supsetEq`	test if first set contains second set

These binary inline operators operate on sets and return a set:

`intersect`	set of elements in both sets
`union`	set of elements in either of the sets

The following unary operator operates on a set and returns a set:

`powerSet`	set of all subsets of a set (including the empty set)

When `S` is a set, then `|S|` denotes the non-negative integer that is the cardinality of `S` (the number of elements in `S`). When `S` and `T` are sets, `S - T` denotes their set difference, the set of elements of `S` that do not occur in `T`.

Examples:

```
find a : bool such that a = (1 in {0,1}) $ true
find b : bool such that b = ({0,1} subset {0,1}) $ false
find c : bool such that c = ({0,1} subsetEq {0,1}) $ true
find d : bool such that d = ({0,1} supset {}) $ true
find e : bool such that e = ({0,1} supsetEq {1,0}) $ true
find A : set of int(0..6) such that A = {1,2,3} intersect {3,4} $ {3}
find B : set of int(0..6) such that B = {1,2,3} union {3,4} $ {1,2,3,4}
find S : set of set of int(0..2) such that S = powerSet({0}) $ {{},{0}}
find x : int(0..9) such that x = |{0,1,2,1,2,1}| $ 3
find T : set of int(0..9) such that T = {0,1,2} - {2,3} $ {0,1}
```

6.9.7 Sequence operators

For two sequences `s` and `t`, `s subsequence t` tests whether the list of values taken by `s` occurs in the same order in the list of values taken by `t`, and `s substring t` tests whether the list of values taken by `s` occurs in the same order and contiguously in the list of values taken by `t`.

When `S` is a sequence, then `|S|` denotes the number of elements in `S`.

```
letting s be sequence(1,1)
letting t be sequence(2,1,3,1)
find a : bool such that s subsequence t $ true
find b : bool such that s substring t $ false
find c : int(1..10) such that c = |t| $ 4
```

6.9.8 Enumerated type operators

`pred`	predecessor of this element in an enumerated type
`succ`	successor of this element in an enumerated type

Enumerated types are ordered, so they support comparisons and the operators *max* and *min*.

```
letting D be new type enum { North, East, South, West }
find a : D such that a = succ(East) $ South
find b : bool such that b = (max([North, South]) > East) $ true
```

The number of elements in an enumerated type can be obtained by using the backtick operator to turn the domain into a list, and then using the cardinality operator:

```
given directions new type enum
letting numberOfDirections be |`directions`|
```

6.9.9 Multiset operators

The following operators take a single argument:

`hist`	histogram of multi-set/matrix
`max`	largest element in ordered set/multi-set/domain/list
`min`	smallest element in ordered set/multi-set/domain/list

The following operator takes two arguments:

`freq`	counts occurrences of element in multi-set/matrix

Examples:

```
letting S be mset(0,1,-1,1)
find x : int(0..1) such that freq(S,x) = 2 $ 1
find y : int(-2..2) such that y = max(S) - min(S) $ 2
find z : int(-2..2) such that z = max([1,2]) $ 2
```

6.9.10 Type conversion operators

`toInt`	maps `true` to 1, `false` to 0
`toMSet`	set/relation/function to multi-set
`toRelation`	function to relation; `function(a --> b)` becomes `relation((a,b))`
`toSet`	multi-set/relation/function to set; `mset(0,0,1)` becomes `{0,1}`

It is currently not possible to use an operator to directly invert `toRelation` or `toSet` when applied to a function, or `toSet` when applied to a relation. By referring to the set of tuples of a function `f` indirectly by means of `toSet(f)`, the set of tuples of a relation `R` by means of `toSet(R)`, or the relation corresponding to a function `g` by `toRelation(g)`, it is possible to use the declarative forms

```
find R : relation of (int(0..1) * int(0..1))
such that toSet(R) = {(0,0),(0,1),(1,1)}

find f : function int(0..1) --> int(0..1)
such that toSet(f) = {(0,0),(1,1)}

find g : function int(0..1) --> int(0..1)
such that toRelation(g) = relation((0,0),(1,1))
```

to indirectly recover the relation or function that corresponds to a set of tuples, or the function that corresponds to a relation. This will fail to yield a solution if a function corresponding to a set of tuples or relation is sought, but that set of tuples or relation does not actually determine a function. An error results if a relation corresponding to a set of tuples is sought, but not all tuples have the same number of elements.

6.9.11 Function operators

`defined`	set of values for which function is defined
`image`	`image(f,x)` is the same as `f(x)`
`imageSet`	`imageSet(f,x)` is `{f(x)}` if `f(x)` is defined, or empty if `f(x)` is not defined
`inverse`	test if two functions are inverses of each other
`preImage`	set of elements mapped by function to an element
`range`	set of values of function
`restrict`	function restricted to a domain

Operators `defined` and `range` yield the sets of values that a function maps between. For all functions `f`, the set `toSet(f)` is contained in the Cartesian product of sets `defined(f)` and `range(f)`.

For a function `f` and a domain `D`, the expression `restrict(f,D)` denotes the function that is defined on the values in `D` for which `f` is defined, and that also coincides with `f` where it is defined.

```
letting f be function(0-->1,3-->4)
letting D be domain int(0,2)
find g : function int(0..4)-->int(0..4) such that
  g = restrict(f, D) $ function(0-->1)
find a : bool such that $ true
  a = ( (defined(g) = defined(f) intersect toSet([i | i : D]))
    /\ (forAll x in defined(g) . g(x) = f(x)) )
```

Applying `image` to values for which the function is not defined may lead to unintended unsatisfiability. The Conjure specific `imageSet` operator is useful for partial functions to avoid unsatisfiability in these cases. The original Essence

definition allows `image` to represent the image of a function with respect to either an element or a set. Conjure does not currently support taking the `image` or `preImage` of a function with respect to a set of elements.

The `inverse` operator tests whether its function arguments are inverses of each other.

```
find a : bool such that a = inverse(function(0-->1),function(1-->0)) $ true
find b : bool such that b = inverse(function(0-->1),function(1-->1)) $ false
```

6.9.12 Matrix operators

The following operator returns a matrix:

`flatten`	list of entries from matrix

`flatten` takes 1 or 2 arguments. With one argument, `flatten` returns a list containing the entries of a matrix with any number of dimensions, listed in the lexicographic order of the tuples of indices specifying each entry. With two arguments `flatten(n,M)`, the first argument `n` is a constant integer that indicates the depth of flattening: the first `n+1` dimensions are flattened into one dimension. Note that `flatten(0,M) = M` always holds. The one-argument form works like an unbounded-depth flattening.

The following operators yield Boolean values:

`allDiff`	test if all entries of a list are different
`alldifferent_except`	test if all entries of a list differ, possibly except value specified in second argument

The following illustrate `allDiff` and `alldifferent_except`:

```
find a : bool such that a = allDiff([1,2,4,1]) $ false
find b : bool such that b = alldifferent_except([1,2,4,1], 1) $ true
```

6.9.13 Partition operators

`apart`	test if a list of elements are not all contained in one part of the partition
`participants`	union of all parts of a partition
`party`	part of partition that contains specified element
`parts`	partition to its set of parts
`together`	test if a list of elements are all in the same part of the partition

Examples:

```
letting P be partition({1,2},{3},{4,5,6})
find a : bool such that a = apart({3,5},P) /\ !together({1,2,5},P) $ true
find b : set of int(1..6) such that b = participants(P) $ {1,2,3,4,5,6}
find c : set of int(1..6) such that c = party(4,P) $ {4,5,6}
find d : bool such that d = ({{1,2},{3},{4,5,6}} = parts(P)) $ true
find e : bool such that e = (together({1,7},P) \/ apart({1,7},P)) $ false
```

These semantics follow the original Essence definition. In contrast, in older versions of Conjure the relationship

```
apart(L,P) = !together(L,P)
```

held for all lists `L` and partitions `P`.

6.9.14 List combining operators

Each of the operators

```
sum product and or xor
```

applies an associative combining operator to elements of a list or set. A list may also be given as a comprehension that specifies the elements of a set or domain that satisfy some conditions.

The following relationships hold for all integers `x` and `y`:

```
sum([x,y]) = (x + y)
product([x,y]) = (x * y)
```

The following relationships hold for all Booleans `a` and `b`:

```
and([a,b]) = (a /\ b)
or([a,b]) = (a \/ b)
xor([a,b]) = ((a \/ b) /\ !(a /\ b))
```

Examples:

```
find x : int(0..9) such that x = sum( {1,2,3} ) $ 6
find y : int(0..9) such that y = product( [1,2,4] ) $ 8
find a : bool such that a = and([xor([true,false]),or([false,true])]) $ true
```

Quantification over a finite set or finite domain of values is supported by `forAll` and `exists`. These quantifiers yield Boolean values and are internally treated as `and` and `or`, respectively, applied to the lists of values corresponding to the set or domain. The following snippets illustrate the use of quantifiers.

```
find a : bool such that a = forAll i in {0,1,2} . i=i*i $ false
find b : bool such that b = exists i : int(0..4) . i*i=i $ true
```

The same variable can be reused for multiple quantifications, as a quantified variable has scope that is local to its quantifier. Older versions of Savile Row do not support using the same name both for quantification and as a global decision variable in a `find`.

An alternative quantifier-like syntax

```
sum i in I . f(i)
```

where `I` is a set, or for domains

```
sum i : int(0..3) . f(i)
```

is supported for the `sum` and `product` operators.

6.9.15 Comprehensions

A list can be constructed by means of a comprehension. A list comprehension is declared by using square brackets `[` and `]` as for other lists, inside which is a generator expression possibly involving some parameter variables, followed by `|`, followed by a comma (`,`) separated sequence of conditions. Each condition is a Boolean expression. The value of a list comprehension is a list containing all the values of the generator expression corresponding to those values of the parameter variables for which all the conditions evaluate to `true`. Comprehension conditions may include `letting` statements, which have local scope within the comprehension.

In a Boolean expression controlling a comprehension, if `L` is a list then `v <- L` behaves similarly to how the expression `v in toMSet(L)` is treated in a quantification. If `c` is a list comprehension, then `|c|` denotes the number of values in `c`.

Examples of list comprehensions:

```
find x : int(0..999) such that x = product( [i-1 | i <- [5,6,7]] ) $ 120
letting M be [1,0,0,1,0]
letting I be domain int(1..5)
find y : int(0..9) such that y = sum( [toInt((i=j) /\ (M[j]>0)) | i : I, j <- M] ) $ 2
find a : bool such that a = and([u<v | (u,v) <- [(0,1),(2**10,2**11),(-1,1)] ]) $ true
find m : int(0..999) such that m = | [M[i] | i : I, M[i] != 0] | $ 2
find n : int(0..999) such that n = | toSet([M[i] | i : I, M[i] != 0]) | $ 1
find b : bool such that b =
    or([ (x=y) | i : I, letting x be i, letting y be M[i] ]) $ true
```

6.9.16 Miscellaneous examples

Some common design patterns are used frequently by experienced modellers.

An often used pattern (also occurring in mixed-integer programming) is to count the number of times a Boolean expression holds across a list. This involves the `toInt` operator used in a quantification or sum. As an example, the following snippet counts the number of entries in a matrix that are each at least as large as the sum of its indices.

```
letting D be domain int(1..3)
letting M be [[5,4,3],[3,4,5],[4,3,5]]
find k : int(1..100) such that
  k = sum i,j : D . toInt(M[i,j] >= i+j) $ 6
```

The `letting` command can be used to define macros to more succinctly express constraints. The final line of the preceding snippet could be replaced by:

```
k = sum( [ toInt(x) : | i, j : D, letting x be (M[i,j] >= i+j) ] )
```

CHAPTER

SEVEN

TUTORIALS

We demonstrate the use of Conjure for some small problems.

7.1 Number puzzle

Authors: András Salamon, Özgür Akgün

We first show how to solve a classic word addition puzzle, due to Dudeney [Dud24]. This is a small toy example, but already illustrates some interesting features of Conjure.

```
   SEND
+  MORE
-------
= MONEY
```

Here each letter represents a numeric digit in an addition, and we are asked to find an assignment of digits to letters so that the number represented by the digits SEND when added to the number represented by MORE yields the number represented by MONEY.

7.1.1 Initial model

We are looking for a mapping (a function) from letters to digits. We can represent the different letters as an enumerated type, with each letter appearing only once.

```
language Essence 1.3
letting letters be new type enum {S,E,N,D,M,O,R,Y}
find f : function letters --> int(0..9)
such that
                   1000 * f(S) + 100 * f(E) + 10 * f(N) + f(D) +
                   1000 * f(M) + 100 * f(O) + 10 * f(R) + f(E) =
    10000 * f(M) + 1000 * f(O) + 100 * f(N) + 10 * f(E) + f(Y)
```

Each Essence specification can optionally contain a declaration of which dialect of Essence it is written in. The current version of Essence is 1.3. We leave out this declaration in the remaining examples.

This model is stored in `sm1.essence`; let's use Conjure to find the solution:

```
conjure solve -ac sm1.essence
```

Unless we specify what to call the solution, it is saved as `sm1.solution`.

```
letting f be function(S --> 0, E --> 0, N --> 0, D --> 0, M --> 0, O --> 0, R --> 0,␣
↪Y --> 0)
```

This is clearly not what we wanted. We haven't specified all the constraints in the problem!

7.1.2 Identifying a missing constraint

In these kinds of puzzles, usually we need each letter to map to a different digit: we need an injective function. Let's replace the line

```
find f : function letters --> int(0..9)
```

by

```
find f : function (injective) letters --> int(0..9)
```

and save the result in file `sm2.essence`. Now let's run Conjure again on the new model:

```
conjure solve -ac sm2.essence
```

This time the solution `sm2.solution` looks more like what we wanted:

```
letting f be function(S --> 2, E --> 8, N --> 1, D --> 7, M --> 0, O --> 3, R --> 6,␣
↪Y --> 5)
```

7.1.3 Final model

There is still something strange with `sm2.essence`. We usually do not allow a number to begin with a zero digit, but the solution maps M to 0. Let's add the missing constraints to file `sm3.essence`:

```
letting letters be new type enum {S,E,N,D,M,O,R,Y}
find f : function (injective) letters --> int(0..9)
such that
                   1000 * f(S) + 100 * f(E) + 10 * f(N) + f(D) +
                   1000 * f(M) + 100 * f(O) + 10 * f(R) + f(E) =
    10000 * f(M) + 1000 * f(O) + 100 * f(N) + 10 * f(E) + f(Y)

such that f(S) > 0, f(M) > 0
```

Let's try again:

```
conjure solve -ac sm3.essence
```

This now leads to the solution we expected:

```
letting f be function(S --> 9, E --> 5, N --> 6, D --> 7, M --> 1, O --> 0, R --> 8,␣
↪Y --> 2)
```

Finally, let's check that there are no more solutions:

```
conjure solve -ac sm3.essence --number-of-solutions=all
```

This confirms that there is indeed only one solution. As an exercise, verify that the first two models have multiple solutions, and that the solution given by the third model is among these. (The first has 1155 solutions, the second 25.)

7.2 The Knapsack problem

Authors: Saad Attieh and Christopher Stone

The Knapsack problem is a classical combinatorial optimisation problem, often used in areas of resource allocation. A basic variant of the Knapsack problem is defined as follows:

- **Given:**
    1. A set of items, each with a weight and a value,
    2. A maximum weight which we call *capacity*,
- **find a set of the items such that**
    1. The sum of the weights of the items in our set is less than or equal to the capacity,and
    2. The sum of the values of the items is maximised.

Informally, think about putting items in a sack such that we maximise the total value of the sack whilst not going over the sack's weight limit.

We begin by showing the entire problem as defined in Essence:

```
given items new type enum
given weight : function (total) items --> int
given gain : function (total) items --> int
given capacity : int
find picked : set of items
maximising sum i in picked . gain(i)
such that (sum i in picked . weight(i)) <= capacity
```

Going through the problem line by line:

We begin by defining the parameters to the problem. Parameters are given in a separate file, allowing different instances of the same problem to be solved without having to change the specification.

Each parameter is denoted with the *given* keyword.

```
given items new type enum
```

This line says that a set of items will be provided in the parameter file as an enum type. Enums are good for labeling items where it makes no sense to attribute a value to each item. So instead of using integers to represent each item, we may just assign names to each item and group the names under an enum type. Below is an example enum declaration, as it would be written in the parameter file:

```
letting items be new type enum {a,b,c,d,e}
```

`a`, `b`, etc. are just names we have given, they could be anything `bread`, `whiskey`, …

```
given weight : function (total) items --> int
```

Another parameter, a function that maps from each item to an integer, we will treat these integers as weights. Since we are describing integers that will be given in the parameter file, no domain (lower/upper bound) is required. Here is an example function parameter as given in a parameter file:

```
letting weight be function
    ( a --> 15
    , b --> 25
    , c --> 45
```

(continues on next page)

(continued from previous page)

```
    , d --> 50
    , e --> 60
    )
```

```
given gain : function (total) items --> int
```

Just the same as the weight parameter, this parameter is used to denote a mapping from each item to a value. An example value for this parameter as it would be defined in the parameter file is:

```
letting gain be function
    ( a --> 10
    , b --> 20
    , c --> 40
    , d --> 40
    , e --> 50
    )
```

The final given:

```
given capacity : int
```

The final parameter – a weight limit. Example value in parameter file:

```
letting capacity be 80
```

```
find picked : set of items
```

The `find` keyword denotes decision variables, these are the variables for which the solver will search for a valid assignment. As is common in Essence problems, our entire problem is modelled using one decision variable named `picked`. Its type is `set of items`; a set of any size whose elements are taken from the `items` domain. Note, the maximum cardinality of the set is implicitly the size of the `items` domain.

```
maximising sum i in picked . gain(i)
```

The `maximising` keyword denotes the objective for the solver; a value for the solver to *maximise*. `minimise` is also a valid objective keyword. The expression `sum i in picked .` is a quantifier. The `sum` says that the values we produce should be summed together. The `i in picked` says we want to list out every element of the set `picked`. The expression given to the `sum` are described by the expression that follows the full-stop (`.`). In this case, we are asking for the image of `i` in the `gain` function. That is, for each item in the set, we are looking up the integer value that the item maps to in the `gain` function and summing these integers.

```
such that (sum i in picked . weight(i)) <= capacity
```

The `such that` keyword denotes a constraint. Here the constraint is formulated in a similar manner to the objective. We are quantifying over the set of chosen items `picked`, looking up the value that the item maps to in the `weights` function and summing these values to together. We enforce that the result of the sum must be less than or equal to the capacity `<= capacity`.

Note that you can post multiple constraints either by using commas between each constraint `,` or by reusing the keyword `such that`.

7.3 Nurse rostering

Authors: András Salamon, Nguyen Dang, Saad Attieh

We now discuss a version of *Nurse Rostering <https://en.wikipedia.org/wiki/Nurse_scheduling_problem>*, a constrained scheduling problem. Variants of this problem are also known by other names, such as *workforce planning* and *staff scheduling*. Unlike versions of this problem studied by operations research practitioners and researchers (such as competition instances [CDDC+19]), we here focus on just a few of the simplest constraints.

Some nurses are available to work in a hospital. Each day is divided into a sequence of shifts, for instance an early-morning shift, a day shift, and a night shift. Each nurse should be assigned to work some shifts during the course of a period of consecutive days. A nurse can be assigned to at most one shift per day. Moreover, for each nurse we need to avoid some forbidden shift patterns within two consecutive days. For example, a nurse cannot work a night shift on one day, and an early-morning shift the next day. We also must make sure to meet the minimum number of nurses required for each shift. These demand values may vary between different days.

7.3.1 Initial specification

To begin with, let's ignore the forbidden patterns, and focus instead on the elements needed to model the problem. We need nurses, shifts for each day, the minimum demand, and a roster.

```
given nNurses, nDays : int(1..)
given shifts new type enum
letting days be domain int(1..nDays)
letting nurses be domain int(1..nNurses)
letting nShifts be |`shifts`|
given forbiddenPatterns : set of tuple (shifts, shifts)
given minimumDemand : function (total) (days, shifts) --> int(0..nNurses)
where
   forAll d : days .
      (sum s : shifts . minimumDemand((d,s))) <= nNurses

find roster: function (days, shifts) --> nurses
$ constraint 1 (Single assignment per day)
$ a nurse can be assigned to at most one shift per day
such that
   forAll nurse : nurses .
      forAll day : days .
         (sum ((d,_),n) in roster . toInt(n=nurse /\ d=day)) <= 1
```

This specification contains the basic elements. We made the choice to use an enumerated type for shifts, positive integers to number the nurses and the days, a set of forbidden patterns (each being a pair of shifts), and a total function mapping each shift slot to a number of nurses to represent the minimum demands. Because we only allow a nurse to work one shift each day, the forbidden patterns can only apply to one day and the next day, with the first shift in a forbidden pattern referring to the first day and the second shift referring to the subsequent day. We also added a `where` condition to ensure that instances are not trivially impossible, by requiring the minimum demand to never exceed the number of nurses.

We also need a test instance. Generating test instances is an interesting subject (for instance, see [AkgunDM+19]) but here we have just made one up.

```
letting nNurses be 5
letting nDays be 7
letting shifts be new type enum {Early, Late, Night}
letting forbiddenPatterns be {
   (Late,Early), (Night,Early), (Night,Late)
```

(continues on next page)

(continued from previous page)

```
}
letting minimumDemand be function (
   (1,Early) --> 2, (1,Late) --> 2, (1,Night) --> 0,
   (2,Early) --> 1, (2,Late) --> 1, (2,Night) --> 2,
   (3,Early) --> 1, (3,Late) --> 1, (3,Night) --> 1,
   (4,Early) --> 0, (4,Late) --> 0, (4,Night) --> 1,
   (5,Early) --> 1, (5,Late) --> 1, (5,Night) --> 2,
   (6,Early) --> 2, (6,Late) --> 1, (6,Night) --> 1,
   (7,Early) --> 0, (7,Late) --> 1, (7,Night) --> 1
)
```

We have 5 nurses, 7 days in the period, three shifts each day, three forbidden pairs of shifts, and some minimum demands for the various shifts.

7.3.2 Changing an overly restrictive assumption

However, on further reflection this first specification is not correct. Since roster is a function `(days, shifts) --> nurses` only one nurse can be assigned to the same shift of the same day. This means that if the minimum demand asks for 2 or more nurses for a particular day and shift, then we can't satisfy this demand. We need to remove the overly restrictive assumption enforced by our choice of representation for `roster`.

Let's change the representation of `roster`, leaving the specification exactly the same up to and including the `where` condition. Instead of mapping each day/shift pair to a single nurse, we could map each day/shift pair to a set of nurses, map each nurse to a set of day/shift pairs, or map each combination of day and nurse to a shift (but with not all combinations needing to be assigned). Each of these choices leads to a slightly different way to model the problem; here we have picked the last. It is left as an exercise to try the others!

```
find roster: function (days, nurses) --> shifts
$ constraint 1 (Single assignment per day)
$ a nurse can be assigned to at most one shift per day
$ NOTE: automatically satisfied because of how "roster" is defined

$ constraint 2 (Under staffing)
$ the number of nurses for each shift suffice for the minimum demand
such that
   forAll day : days .
      forAll shift : shifts .
         (sum ((d,_),s) in roster . toInt(d=day /\ s=shift))
            >= minimumDemand((day,shift))

$ constraint 3 (Shift type successions)
$ the shift type assignments of one nurse on two consecutive days
$ must not violate any forbidden succession
such that
   forAll d : int(1..(nDays-1)) .
      forAll n : nurses .
         !((roster(d,n), roster(d+1,n)) in forbiddenPatterns)
```

Note that in this specification, the first constraint is automatically satisfied because of the way we have defined `roster` as a function from a day/nurse pair to a shift. So changing the representation of `roster` has not only removed the overly restrictive assumption that only one nurse can be assigned to a day/shift pair, but also dealt with the first real constraint.

We have added a second constraint to enforce the minimum demand for each shift, by requiring that the number of nurses mapped to each day/shift pair is at least as large as the minimum demand for that day/shift pair.

Finally, we have added a third constraint to ensure that forbidden shift patterns do not occur.

7.3.3 Final model

Unfortunately, the second specification is not accepted by Conjure. The `roster` function is expecting a single pair as its argument, but we have given two arguments (a day and a nurse). We replace the last constraint by a version that corrects this syntax error:

```
$ constraint 3 (Shift type successions)
$ the shift type assignments of one nurse on two consecutive days
$ must not violate any forbidden succession
such that
   forAll d : int(1..(nDays-1)) .
      forAll n : nurses .
         !((roster((d,n)), roster((d+1,n))) in forbiddenPatterns)
```

This is a specification that is acceptable to Conjure and which captures the key constraints we wanted to include.

Assuming that the third specification is in file `model3.essence` and the test instance in file `test.param`, we can run Conjure to solve the instance.

```
conjure solve -ac model3.essence test.param
```

Without any specification, the default solver is Minion [CDDC+19], a constraint programming solver. After quite some time, this creates the following solution:

```
letting roster be function(
   (1, 2) --> Early, (1, 3) --> Early, (1, 4) --> Late,  (1, 5) --> Late,
   (2, 2) --> Early, (2, 3) --> Late,  (2, 4) --> Night, (2, 5) --> Night,
   (3, 2) --> Early, (3, 3) --> Late,  (3, 4) --> Night, (4, 5) --> Night,
   (5, 2) --> Early, (5, 3) --> Late,  (5, 4) --> Night, (5, 5) --> Night,
   (6, 1) --> Early, (6, 2) --> Early, (6, 3) --> Late,  (6, 4) --> Night,
   (7, 4) --> Night, (7, 5) --> Late)
```

A much faster way to obtain a solution is to ask Minion to use the `domoverwdeg` variable ordering, which is often effective on constrained scheduling problems:

```
conjure solve -ac --solver-options='-varorder domoverwdeg' model3.essence test.param
```

Choosing the right parameters to control solver behaviour is important but not generally well understood, and we leave discussion of this problem for another time.

7.4 Labelled connected graphs

Author: András Salamon (with thanks to Roy Dyckhoff for proofreading)

We now illustrate the use of Conjure for a more realistic modelling task, to enumerate all labelled connected graphs. The number of labelled connected graphs over a fixed set of n distinct labels grows quickly; this is OEIS sequence A001187.

We first need to decide how to represent graphs. A standard representation is to list the edges. One natural representation for each edge is as a set of two distinct vertices. Vertices of the graph are labelled with integers between 1 and n, and each vertex is regarded as part of the graph, whether there is some edge involving that vertex or not.

```
letting n be 4
letting G be {{1,2},{2,3},{3,4}}
```

In this specification, we declare two aliases. The number of vertices n is first defined as 4. Then G is defined as a set of edges.

This specification is saved in a file `path-4.param` that we refer to later. We should also have a different graph that is not connected:

```
letting n be 4
letting G be {{1,2},{4,3}}
```

which is saved in file `disconnected-4.param`.

We now need to express what it means for a graph to be connected.

7.4.1 Model 1: distance matrix

In our first attempt, we use a matrix of distances. Each entry `reach[u,v]` represents the length of a shortest path from u to v, or n if there is no path from u to v. To enforce this property, we use several constraints, one for each possible length; there are four ranges of values we need to cover. A distance of 0 happens when u and v are the same vertex. A distance of 1 happens when there is an edge from u to v. When the distance is greater than 1 but less than n, then there must be some vertex that is a neighbour of u from which v is reachable in one less step. Finally, the distance of n is used when no neighbour of u can reach v (and in this case, the neighbours all have distance of n to v as well).

```
given n : int(1..)
letting vertices be domain int(1..n)
given G : set of set (size 2) of vertices
find reach : matrix indexed by [vertices, vertices] of int(0..n)
such that
  forAll u,v : vertices .
     ((reach[u,v] = 0) -> (u=v))
  /\ ((reach[u,v] = 1) -> ({u,v} in G))
  /\ (((reach[u,v] > 1) \/ (reach[u,v] < n)) ->
      (exists w : vertices . ({u,w} in G) /\ (reach[w,v] = reach[u,v] - 1)))
  /\ ((reach[u,v] = n) -> (forAll w : vertices . !({u,w} in G) \/ (reach[w,v] = n)))
find connected : bool
such that
  connected = (forAll u,v : vertices . reach[u,v] < n)
```

This is stored in file `gc1.essence`. The values of n and G will be specified later as parameters, such as via the `path-4.param` or `disconnected-4.param` files.

In the model, first the matrix `reach` is specified by imposing the four conditions that we mentioned. Finally a Boolean variable is used to conveniently indicate whether the `reach` matrix represents a connected graph or not; in a connected graph every vertex is reachable from every other vertex.

Let's now try this model with the two graphs defined so far.

```
conjure solve -ac gc1.essence path-4.param
conjure solve -ac gc1.essence disconnected-4.param
```

In the solutions found by Conjure, the matrix `reach` indicates the distances between each pair of vertices. In the solution for the connected graph `gc1-path-4.solution` all entries are at most 3.

```
letting connected be true
letting reach be
  [[0, 1, 2, 3; int(1..4)], [1, 0, 1, 2; int(1..4)],
   [2, 1, 0, 1; int(1..4)], [3, 2, 1, 0; int(1..4)]; int(1..4)]
$ Visualisation for reach
$ 0 1 2 3
$ 1 0 1 2
```

(continues on next page)

(continued from previous page)

```
$ 2 1 0 1
$ 3 2 1 0
```

In contrast, in the solution for the disconnected graph `gc1-disconnected-4.solution` there are some entries that are 4:

```
letting connected be false
letting reach be
  [[0, 1, 4, 4; int(1..4)], [1, 0, 4, 4; int(1..4)],
   [4, 4, 0, 1; int(1..4)], [4, 4, 1, 0; int(1..4)]; int(1..4)]
$ Visualisation for reach
$ 0 1 4 4
$ 1 0 4 4
$ 4 4 0 1
$ 4 4 1 0
```

Graphs with four vertices are good for quick testing but are too small to notice much difference between models. Small differences are important for tasks such as enumerating many objects, when even a small difference is multiplied by the number of objects. For testing we can create other parameter files containing graphs with more vertices. Notice that we do not have to change the model, only the parameter files containing the input data.

Testing with larger graphs of say 1000 vertices, it becomes clear that this first model works but does not scale well. It computes the lengths of the shortest paths between pairs of vertices, from which we can deduce whether the graph is connected. This is quite round-about! We can now try to improve the model by asking the system to do less work. After all, we don't actually need all the pairwise distances.

7.4.2 Model 2: reachability matrix

In the following model, stored as file `gc2.essence`, the reachability matrix uses Boolean values for the distances rather than integers, with `true` representing reachable and `false` unreachable. Each entry `reach[u,v]` represents whether it is possible to reach v by some path that starts at u. This is modelled as the disjunction of three conditions: u is reachable from itself, any neighbour of u is reachable from it, and if v is not a neighbour of u then there should be a neighbour w of u so that v is reachable from w.

```
given n : int(1..)
letting vertices be domain int(1..n)
given G : set of set (size 2) of vertices
find reach : matrix indexed by [vertices, vertices] of bool
such that
  forAll u,v : vertices . reach[u,v] =
    ((u = v) \/ ({u,v} in G) \/
    (exists w : vertices . ({u,w} in G) /\ reach[w,v]))
find connected : bool
such that
  connected = (forAll u,v : vertices . reach[u,v])
```

In the solutions found by Conjure, the reachability matrix contains regions of true entries indicating the connected components.

In the connected graph all entries are true:

```
letting connected be true
letting reach be
  [[true, true, true, true; int(1..4)], [true, true, true, true; int(1..4)],
   [true, true, true, true; int(1..4)], [true, true, true, true; int(1..4)];
```

(continues on next page)

(continued from previous page)

```
   int(1..4)]
$ Visualisation for reach
$ T T T T
$ T T T T
$ T T T T
$ T T T T
```

In contrast, in the disconnected graph there are some false entries:

```
letting connected be false
letting reach be
  [[true, true, false, false; int(1..4)], [true, true, false, false; int(1..4)],
   [false, false, true, true; int(1..4)], [false, false, true, true; int(1..4)];
   int(1..4)]
$ Visualisation for reach
$ T T _ _
$ T T _ _
$ _ _ T T
$ _ _ T T
```

This model takes about half as long as the previous one, but is still rather slow for large graphs.

7.4.3 Model 3: structured reachability matrices

In the previous two models the solver may spend a long time early in the search process looking for ways to reach vertices that are far away, even though it would be more efficient to focus the early stages of search on vertices close by. It is possible to improve performance by guiding the search to consider nearby vertices before vertices that are far from each other. The following model `gc3.essence` uses additional decision variables to more precisely control how the desired reachability matrix should be computed. There are multiple reachability matrices. Each corresponds to a specific maximum distance. The first n by n matrix `reach[0]` expresses reachability in one step, and is simply the adjacency matrix of the graph. The entry `reach[k,u,v]` expresses whether v is reachable from u via a path of length at most 2**k. If a vertex v is reachable from some vertex u, then it can be reached in at most n-1 steps. (Note: in this model a vertex cannot reach itself in zero steps, so a graph with a single vertex is not regarded as connected.)

```
given n : int(1..)
letting vertices be domain int(1..n)
given G : set of set (size 2) of vertices
letting m be sum([1 | i : int(0..64), 2**i <= n])
find reach : matrix indexed by [int(0..m), vertices, vertices] of bool
such that
  forAll u,v : vertices . reach[0,u,v] = ({u,v} in G),
  forAll i : int(0..(m-1)) . forAll u,v : vertices . reach[i+1,u,v] =
    (reach[i,u,v] \/ (exists w : vertices . (reach[i,u,w] /\ reach[i,w,v]))),
find connected : bool
such that
  connected = (forAll u,v : vertices . reach[m,u,v])
```

The variable m is used to compute the number of matrices that are required; this is the smallest integer that is not less than the base-2 logarithm of n. (This is computed by discrete integration as Conjure currently does not support a logarithm operator; this may change in a future release.) The value of `connected` is then based on whether whether `reach[m]` contains any false entries.

This model is the fastest yet, but it generates intermediate distance matrices, each containing n**2 variables. We omit the solutions here, but they show how the number of true values increases, until reaching a fixed point.

7.4.4 Model 4: connected component

Each of the three models so far deals with all possible pairs of vertices. The number of possible pairs of vertices is quadratic in the number of vertices. However, many graphs are sparse, with a number of edges that is bounded by a linear function of the number of vertices. For sparse graphs, and especially those with many vertices, it is therefore important to only consider the edges that are present rather than all possible pairs of vertices. The next model `gc4.essence` uses this insight, and is indeed faster than any of the three previous ones.

The model builds on the fact that a graph is disconnected if, and only if, its vertices can be partitioned into two sets, with no edges between vertices in the two different sets. Here C is used to indicate a subset of the vertices. There are three constraints. The first is that C must contain some vertex. The second is that C must be a connected component; each vertex in C is connected to some other vertex in C (unless C only contains a single vertex). The third is that the value of `connected` is determined by whether it is possible to find some vertex that is not in C. The following is an attempt to capture these constraints in an Essence specification.

```
given n : int(1..)
letting vertices be domain int(1..n)
given G : set of set (size 2) of vertices
find C : set of vertices
find connected : bool
such that
  exists u : vertices . u in C,
  forAll e in G . (min(e) in C) = (max(e) in C),
  connected = !(exists u : vertices . !(u in C))
```

This is the solution for `disconnected-4.param`:

```
letting C be {1, 2}
letting connected be false
```

Model `gc4.essence` yields a solution quickly. Unfortunately it can also give incorrect results: letting C be the set of all vertices and letting `connected` be true is always a solution, whether the graph is connected or not. This can be confirmed by asking Conjure to generate all solutions:

```
conjure solve -ac --number-of-solutions=all gc4.essence
```

This gives two solutions, the one above and the following one:

```
letting C be {1, 2, 3, 4}
letting connected be true
```

It is actually possible to ensure that this "solution" is never the first one generated, and then to ask Conjure to only look for the first solution; if the graph is not connected then the first solution will correctly indicate its status. However, this relies on precise knowledge of the ordering heuristics being employed at each stage of the toolchain.

The problem with this fourth specification is that it only captures the property that C is a union of connected components. We would need to add additional constraints to enforce the property that C should contain only one connected component. This can be done, but is not especially efficent.

7.4.5 Model 5: minimal connected component

Let's look for a robust approach that won't unexpectedly fail if parts of the toolchain change which optimisations they perform or the order in which evaluations occur.

One option could be to look for solutions of a more restrictive model which includes an additional constraint that requires some vertex to not be in C. This model would have a solution precisely if the graph is *not* connected. Failure to find solutions to this model would then indicate connectivity. It is possible to call Conjure from a script that uses the failure to find solutions to conclude connectivity, but the Conjure toolchain currently does not support testing for the presence of solutions directly.

In place of the missing "if-has-solution" directive, we could instead quantify over all possible subsets of vertices. Such an approach quickly becomes infeasible as n grows (and is much worse than the models considered so far), because it attempts to check 2**n subsets.

As another option, we can make use of the optimisation features of Essence to find a solution with a C of minimal cardinality. This ensures that C can only contain one connected component. Choosing a minimal C ensures that when there is more than one solution, then the one that is generated always indicates the failure of connectivity. Since we don't care about the minimal C, as long as it is smaller than the set of all vertices if possible, we also replace the general requirement for non-emptiness by a constraint that always forces the set C to contain the vertex labelled 1.

```
given n : int(1..)
letting vertices be domain int(1..n)
given G : set of set (size 2) of vertices
find C : set of vertices
find connected : bool
such that
  1 in C,
  forAll e in G . (min(e) in C) = (max(e) in C)
minimising |C|
```

This model `gc5.essence` is still straightforward, even with the additional complication to rule out incorrect solutions. Out of the correct models so far, this tends to generate the smallest input files for the back-end constraint or SAT solver, and also tends to be the fastest.

7.4.6 Generating all connected graphs

We now have a fast model for graph connectivity. Let's modify it as `gce1.essence`, hardcoding n to be 4 and asking the solver to find G as well as C.

```
letting n be 4
letting vertices be domain int(1..n)
find G : set of set (size 2) of vertices
find C : set of vertices
such that
  1 in C,
  forAll e in G . (min(e) in C) = (max(e) in C)
minimising |C|
```

We now ask for all solutions:

```
conjure solve -ac --number-of-solutions=all gce1.essence
```

However, this finds only one solution!

The solver finds one solution that minimises `|C|`; this minimisation is performed globally over all possible solutions. This is what we intended when G was given, but is not what we want if our goal is to generate *all* connected graphs. We want

to minimise C for each choice of G, producing one solution for each G. Currently there is no way to tell Conjure that minimisation should be restricted to the decision variable C.

Checking whether there is a nontrivial connected component seems to be the most efficient model for graph connectivity, but it doesn't work in the setting of generating all connected graphs. We therefore need to choose one of the other models to start with, say the iterated adjacency matrix representation.

We now use this model of connectivity to enumerate the labelled connected graphs over the vertices `{1,2,3,4}`. Previously we checked connectivity of a given graph G. We now instead ask the solver to find G, specifying that it be connected. We do this by asking for the same adjacency matrix `reach` as before, but in addition asking for the graph G. We also hardcode n, so no parameter file is needed, and add the condition that previously determined the value of the `connected` decision variable as a constraint.

```
letting n be 4
letting vertices be domain int(1..n)
find G : set of set (size 2) of vertices
letting m be sum([1 | i : int(0..64), 2**i <= n])
find reach : matrix indexed by [int(0..m), vertices, vertices] of bool
such that
  forAll u,v : vertices . reach[0,u,v] = ({u,v} in G),
  forAll i : int(0..(m-1)) . forAll u,v : vertices . reach[i+1,u,v] =
    (reach[i,u,v] \/ (exists w : vertices . (reach[i,u,w] /\ reach[i,w,v]))),
  forAll u,v : vertices . reach[m,u,v]
```

If this model is in the file `gce2.essence`, then we now need to explicitly ask Conjure to generate all the possible graphs:

```
conjure solve -ac --number-of-solutions=all gce2.essence
```

In this case Conjure generates 38 solutions, one solution per file.

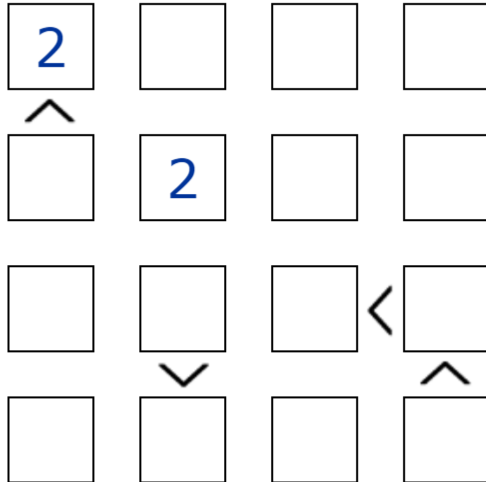
Instead of listing the edges of a graph, and then deriving the adjacency matrix as necessary, it is also possible to use the adjacency matrix representation. As an exercise, modify the models of connectivity to use the adjacency matrix representation instead of the set of edges representation.

7.5 Futoshiki

Authors: Ruth Hoffmann and Gökberk Koçak

7.5.1 Problem

`n x n` board where each column and row is filled with the unique numbers from 1 to `n`, similar to a sudoku. In contrast to sudoku, there are less than and greater than symbols between cells indicating that one cell has to be filled with a number greater than (or less than) than the cell on the other side of the operator.



7.5.2 Essence Model

Let us look at the model first.

```
language Essence 1.3
given n : int
letting DOMAIN be domain int(1..n)
given hints : function (DOMAIN, DOMAIN) --> DOMAIN
given less_than : relation of ((DOMAIN, DOMAIN) * (DOMAIN, DOMAIN))
find board : matrix indexed by [DOMAIN, DOMAIN] of DOMAIN
such that
    forAll (hint,num) in hints .
        board[hint[1], hint[2]] = num,
    forAll i: DOMAIN .
        allDiff(board[i,..]),
    forAll j: DOMAIN .
        allDiff(board[..,j]),
    forAll (l,g) in less_than .
        board[l[1],l[2]] < board[g[1],g[2]]
```

The line by line explanation starts here.

```
given n : int
```

Defines the size of the matrix and the maximal number of elements that we will add.

```
letting DOMAIN be domain int(1..n)
```

We start at 1 and go up to `n` (for both the elements of the cells and the cell locations).

```
given hints : function (DOMAIN, DOMAIN) --> DOMAIN
```

Here we define which cells are already filled in using a function. We map the coordinates onto the number that is in that cell. It is important to notice that functions in essence are partial functions not total. This means that not everything gets mapped.

```
given less_than : relation of ((DOMAIN, DOMAIN) * (DOMAIN, DOMAIN))
```

Here we define where the relation symbols are placed, it depends on which way you read the symbol we are "only" modelling less than. So should there be a "greater than" signs between two cells, say `(2,1)` and `(1,1)` then we would order them as `(1,1)` is less than `(2,1)`.

```
find board : matrix indexed by [DOMAIN, DOMAIN] of DOMAIN
```

We are now telling the solver that we are trying to find a `n x n` board with elements from 1 to `n` in each cell.

```
such that
```

This is the beginning of the constraints block.

```
forAll (hint,num) in hints .
   board[hint[1], hint[2]] = num,
```

This constraint defines the hints, so the cells that are filled in when we get the puzzle.

```
forAll i: DOMAIN .
    allDiff(board[i,..]),
```

This constraint defines that every cell in a row has to be a unique number between 1 and n.

```
forAll j: DOMAIN .
    allDiff(board[..,j]),
```

This constraint defines that every cell in a column has to be a unique number between 1 and n.

```
forAll (l,g) in less_than .
    board[l[1],l[2]] < board[g[1],g[2]]
```

Finally this constraint enforces the less than relation. `l` is the number that is the cell that contains the number that is less than then the cell `g`.

7.5.3 Instance

We save the instance in a `.essence-param` file.

```
letting n be 4
letting hints be function(
        (1,1) --> 2,
        (2,2) --> 2
)
letting less_than be relation(
        ((1,1) , (2,1)),
        ((4,2) , (3,2)),
        ((3,3) , (3,4)),
        ((3,4) , (4,4))
)
```

The `.essence-param` file contains the information about our starting board of a specific instance that we want to solve. See the picture at the beginning to see what it looks like.

```
letting n be 4
```

We are dealing with a 4 by 4 board.

```
letting hints be function(
        (1,1) --> 2,
        (2,2) --> 2
)
```

There will be two 2 s on the board given as a hint. One in the top left corner (1,1) and the second number 2 in cell (2,2).

```
letting less_than be relation(
        ((1,1) , (2,1)),
        ((4,2) , (3,2)),
        ((3,3) , (3,4)),
        ((3,4) , (4,4))
)
```

There are 4 relation symbols on the board, between cells.

7.5.4 Solving

Using the ESSENCE pipeline, we can solve our sample instance by typing the following:

```
conjure solve futoshiki-model.essence futoshiki-instance.essence-param
```

The result will be saved into a .solution file which will look something like this:

```
letting board be
        [[2, 1, 4, 3; int(1..4)], [4, 2, 3, 1; int(1..4)], [3, 4, 1, 2; int(1..4)],␣
↪[1, 3, 2, 4; int(1..4)]; int(1..4)]
$ Visualisation for board
$ 2 1 4 3
$ 4 2 3 1
$ 3 4 1 2
$ 1 3 2 4
```

7.6 BIBD

Authors: Chris Jefferson and Alice Lynch

This tutorial discusses a classic constraint problem and introduces the use of quantifiers in Essence.

7.6.1 The Problem

Balanced Incomplete Block Design (BIBD) is a problem from the field of experimental design. It is best explained with an example.

Emily wants to establish which crops (potato,corn,broccoli,carrot,cucumber, tomato) grow best in Scotland. She has recruited 4 farmers who are happy to help by growing some of the crops. Unfortunately none of the farmers have enough space to grow every crop, they can each grow 3 different crops. Emily is concerned that the different environment of each farm may impact the crops growth. Therefore she wants to make sure that each farmer grows a different combination of crops and that every crop has been grown in the same number of different farms. This approach is called Balanced Incomplete Block Design (BIBD).

We can build a model to tell us the crops that each farm should grow.

7.6.2 The Model

We need to specify the crops, the number of farms, the number of crops that can be grown per farm, the number of different farms that will grow each crop and the number of crops each pair of farmers has in common.

Emily has decided that she wants each crop to be grown in 2 different farms, and that each pair of farmers will have 1 crop in common.

Essence will take a .param file containing the values of the initial parameters. In the .param file we should define the parameters:

```
letting crops be new type enum {potato,corn,broccoli,carrot,cucumber, tomato}
letting farms be 4
letting crops_per_farm be 3
letting farms_per_crop be 2
letting overlap be 1
```

The model will be in a .essence file. It should start by accessing the provided parameters, this uses the given keyword, followed by the names of the parameters and their type.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum
```

Next, we need to define what we are looking for. The 'find' keyword indicates that the solver should find a value to for that variable. We want to find a set containing sets of crops. Each set of crops is a crop assignment for a farm.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops
```

Once the parameters and ***targets?** of the model have been defined, we should define the constraints. `such that` indicates the start of the constraints.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that
```

Result:

```
{}
```

With no constraints it produces an empty set for crop assignment.

The first, basic, constraints is the number of farms. The number of sets in the crop_assignment set should equal the numbers of farms. `|crop_assignment|` indicates the size of the crop_assignment set. By setting the size equal to the number of farms (after the such that keyword) the solver will only produce solutions where the size of the set is the same as the number of farms. A comma on the end of line indicates that there are more constraints to follow.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that
```

(continues on next page)

(continued from previous page)

```
|crop_assignment| = farms,
```

Result:

```
{{},
 {cucumber},
 {cucumber, tomato},
 {tomato}}
```

The model now produces four 'farms' but the number of crops assigned to each are not suitable.

Next we want to apply the number of crops per farm constraint to every set in the crop assignment set. The `forAll` keyword will apply the constraint (`|farm| = crops_per_farm`) across every element in the crop_assignment set (represented by `farm`). The `.` separates the constraint from the quantifier setup.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that

|crop_assignment| = farms,
forAll farm in crop_assignment . |farm| = crops_per_farm,
```

Result:

```
{{broccoli, carrot, cucumber},
 {broccoli, carrot, tomato},
 {broccoli, cucumber, tomato},
 {carrot, cucumber, tomato}}
```

The model now has the correct number of farms and assigns the correct number of crops per farms, but doesn't assign all types of crops.

The next constraint is number of farms with a given crop. This is more complex than the previous constraints. Let's go over it step by step. For every crop we need to find the number of farms assigned that crop and set it to equal the parameter Emily chose for farms per crop. In order to find this we first use a `forAll` to apply the constraint to every crop. `forAll crop : crops . [OurCalculation] = farms_per_crop`

Then we need to count every farm that is planting that crop. For this we should use the `sum` quantifier rather than the `forAll`(`sum farm in crop_assignment . [Action]`). `sum` will add together all the results of the chosen action. In order to use sum to count the number of farms that contain a crop we need to return 1 if the farm is planting the crop and 0 otherwise. The `in` keyword can be used to check if a crop is present in a farm, the resulting boolean can be converted to 1 or 0 using `toInt`.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that

|crop_assignment| = farms,
forAll farm in crop_assignment . |farm| = crops_per_farm,
```

(continues on next page)

(continued from previous page)

```
forAll crop : crops . (sum farm in crop_assignment . toInt(crop in farm)) = farms_per_
↪crop,
```

Result:

```
{{potato, carrot, tomato},
 {potato, cucumber, tomato},
 {corn, broccoli, carrot},
 {corn, broccoli, cucumber}}
```

Our model now produces a crop assignment that assigns the correct number of crops to each farmer and the correct number of crops in total but there is lot of overlap between the first and second farmer and between the third and fourth farmer but very little overlap between the two pairs. This is why Emily specified the overlap constraint (sometimes called lambda in BIBD models). In order to make sure that every pair of farmers have at least 1 crop in common we need to define another constraint.

We need to check every pair of farms, we can do this by using two `forAll` keywords (`forAll farm1 in crop_assignment. forAll farm2 in crop_assignment . [OurConstraint]`). We can then use the `intersect` keyword to get all crops that the two farms have in common. The `||` notation can be used to get the size of the intersect which we can then set equal to the overlap parameter (`|farm1 intersect farm2| = overlap`).

However, running the model at this point produces no solutions, as iterating over the crop_assignment in this way means that sometimes farm1 and farm2 will be the same farm, so the intersection will be the number of crops assigned to the farm (3) and never be 1 (the overlap parameter), resulting in no valid solutions.

In order to avoid this we need to add an further condition to the constraint which checks they are not the same farm before applying the constraint. `->` is used, where the left hand side has a condition and the right hand side has a constraint which is only used if the left hand side is true. `farm1 != farm2 -> |farm1 intersect farm2| = overlap`

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that

|crop_assignment| = farms,
forAll farm in crop_assignment . |farm| = crops_per_farm,
forAll crop : crops . (sum farm in crop_assignment . toInt(crop in farm)) = farms_per_
↪crop,
forAll farm1 in crop_assignment. forAll farm2 in crop_assignment . farm1 != farm2 ->␣
↪|farm1 intersect farm2| = overlap
```

Result:

```
{{potato, broccoli, tomato},
 {potato, carrot, cucumber},
 {corn, broccoli, cucumber},
 {corn, carrot, tomato}}
```

This model produces a valid solution!

7.6.3 Improvements

Our model now works and produces a correct solution but the code could be improved in places.

There is a nicer way to do the final constraint, instead of using a second `forAll` we can use `{farm1, farm2}` and `subsetEq` to generate all pairs that can be made up from a given set.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set of set of crops

such that

|crop_assignment| = farms,
forAll farm in crop_assignment . |farm| = crops_per_farm,
forAll crop : crops . (sum farm in crop_assignment . toInt(crop in farm)) = farms_per_
↪crop,
forAll {farm1, farm2} subsetEq crop_assignment . |farm1 intersect farm2| = overlap
```

Providing information in the find statements rather than as constraints often leads to better perform. Essence provides attributes which can be attached to find statements . One of them is size k, which tells Essence that a set is of size k. In our model the number of farms and the number of crops per farm are in effect the size of the crop_assignment set and the size of the sets within the crop_assignment set. Therefore we can move these definitions out of the list of constraints and into the find statement.

```
given farms, crops_per_farm, farms_per_crop, overlap: int
given crops new type enum

find crop_assignment: set (size farms) of set (size crops_per_farm) of crops

such that
forAll crop : crops . (sum farm in crop_assignment . toInt(crop in farm)) = farms_per_
↪crop,
forAll {farm1, farm2} subsetEq crop_assignment . |farm1 intersect farm2| = overlap
```

7.7 Semigroups, Monoids and Groups

Authors: Chris Jefferson and Alice Lynch

This tutorial discusses how to model semigroups, monoids and groups in Essence.

7.7.1 The Problem

Semigroups, monoids and groups are all examples of binary operations, with added conditions.

We will begin by building a binary operation. A binary relation `R` on a domain `S` is a two-argument function which maps two elements of `S` to a third element of `S`. We will make `S` be the integers from `1` to `n`, for some given `n`.

```
given n : int
letting S be domain int(1..n)

find R : function(total) (S,S) --> S
```

We make a new type of size `n` to represent the set the operation is defined on. We then define a function from `(S,S)` to `S`. Technically, this function doesn't take two arguments - it takes a single argument which is a pair of values from `S`. This is mathematically the same, but will change how we use `R`.

We will begin by creating a solution to this, for `n = 4`.

Result:

```
letting R be
      function((1, 1) --> 1, (1, 2) --> 1, (1, 3) --> 1, (1, 4) --> 1,
               (2, 1) --> 1, (2, 2) --> 1, (2, 3) --> 1, (2, 4) --> 1,
               (3, 1) --> 1, (3, 2) --> 1, (3, 3) --> 1, (3, 4) --> 1,
               (4, 1) --> 1, (4, 2) --> 1, (4, 3) --> 1, (4, 4) --> 1)
```

At the moment this is quite boring, as the function can take any value at all! Asking Conjure how many solutions this problem has is unreasonable, but we can figure it out with maths: $$4^{16} = 4,294,967,296$$. Let's try adding some constraints.

7.7.2 Semigroups

The simplest object we will consider is a **semigroup**. A semigroup adds one constraint to our binary operation, **associativity**. A binary operation is associative if for all i,j and k in S, R(i,R(j,k)) = R((R(i,j),k). This might look very abstract, but it is true of many binary operations, for example given integers i,j and k, (i+j)+k = i+(j+k), and (i * j) * k = i * (j * k).

We begin by saying we want to check `forAll i,j,k: S`. The strangest bit is all of the brackets seem doubled. Your vision isn't failing, this is because `M` is a one argument function (and we use `M(x)` to apply `M` to `x`), but `M` takes a tuple as its argument (which we write as `(i,j)`), so to apply `M` to `i` and `j` we write `M((i,j))`.

```
given n : int
letting S be domain int(1..n)

find R : function(total) (S,S) --> S

such that

forAll i,j,k: S. R((i,R((j,k)))) = R((R((i,j)),k))
```

Result:

```
letting R be
      function((1, 1) --> 1, (1, 2) --> 1, (1, 3) --> 1, (1, 4) --> 1,
               (2, 1) --> 1, (2, 2) --> 1, (2, 3) --> 1, (2, 4) --> 1,
               (3, 1) --> 1, (3, 2) --> 1, (3, 3) --> 1, (3, 4) --> 1,
               (4, 1) --> 1, (4, 2) --> 1, (4, 3) --> 1, (4, 4) --> 1)
```

The first result is still the same, but there are fewer solutions to be found now - only 3,492. Is this correct? It's always good to check. This number was first published in 1955, by George E. Forsythe, in his paper "SWAC Computes 126 Distinct Semigroups of Order 4". Where does the number 126 come from? This small number comes from ignoring cases where the semigroup is the same except for rearranging the numbers 1,2,3,4. The number we found, 3,492, is found in the paper.

7.7.3 Monoids

Let's move further to monoids. A monoid is a semigroup with an extra condition, there has to exist some element of the semigroup, which we will call *e*, which acts as an **identity**. An **identity** is an element such that for all `i` in `S`, `R(e,i) = R(i,e) = e`.

Firstly we will add a variable to store the value of this `e`, and then add the extra constraint which makes it an identity:

```
given n : int

letting S be domain int(1..n)

find R : function (total) (S,S) --> S

find e : S

such that

forAll i,j,k: S. R((i,R((j,k)))) = R((R((i,j)),k)),
forAll i : S. M((e,i)) = i /\ M((i,e)) = i,
```

Result:

```
letting R be
      function((1, 1) --> 1, (1, 2) --> 1, (1, 3) --> 1, (1, 4) --> 1,
               (2, 1) --> 1, (2, 2) --> 1, (2, 3) --> 1, (2, 4) --> 2,
               (3, 1) --> 1, (3, 2) --> 1, (3, 3) --> 1, (3, 4) --> 3,
               (4, 1) --> 1, (4, 2) --> 2, (4, 3) --> 3, (4, 4) --> 4)
letting e be 4
```

We now have only 624 solutions! We can check this by looking at the amazing online encyclopedia of integer sequences https://oeis.org/A058153 , which tells us there are indeed 624 "labelled monoids" of order n.

7.7.4 Groups

Finally, let us move to groups. Groups add one important requirement, the concept of an **inverse**. Given some `i` in `S`, `j` is an inverse of `i` if `R((i,j)) = R((j,i)) = e`, where `e` is our already existing identity.

We will store the inverses as an extra array, and then add this final constraint:

```
given n : int

letting S be domain int(1..n)

find R : function (total) (S,S) --> S

find e : S

find inv: function S --> S

such that

forAll i,j,k: S. R((i,R((j,k)))) = R((R((i,j)),k)),
forAll i : S. R((e,i)) = i /\ R((i,e)) = i,

forAll i : S. R((i,inv(i))) = e /\ R((inv(i),i)) = e
```

Result:

```
letting R be
      function((1, 1) --> 1, (1, 2) --> 2, (1, 3) --> 3, (1, 4) --> 4,
               (2, 1) --> 2, (2, 2) --> 1, (2, 3) --> 4, (2, 4) --> 3,
               (3, 1) --> 3, (3, 2) --> 4, (3, 3) --> 1, (3, 4) --> 2,
               (4, 1) --> 4, (4, 2) --> 3, (4, 3) --> 2, (4, 4) --> 1)
letting e be 4
letting inv be function(1 --> 1, 2 --> 2, 3 --> 3, 4 --> 4)
```

This solution has much more going on than our previous ones! For example, each row and column contains the numbers from `1` to `4`, in some order. This (and many, many other results) are true for all groups (but we won't prove this here!). This problem only has 16 solutions, and once we removed the groups which are made by just swapping around 1,2,3 and 4, we would find there was only 2 groups! The extra structure means there are only a small number of groups for each size, compared to the number of semigroups and monoids.

There are many special types of groups; we will consider just one here, **abelian** groups. A group is **abelian** if for all `i` and `j` in `S`, `R((i,j)) = R((j,i))`. Let's add this condition!

```
given n : int

letting S be domain int(1..n)

find R : function (total) (S,S) --> S

find e : S

find inv: function S --> S

such that

forAll i,j,k: S. R((i,R((j,k)))) = R((R((i,j)),k)),
forAll i : S. R((e,i)) = i /\ R((i,e)) = i,
forAll i : S. R((i,inv(i))) = e /\ R((inv(i),i)) = e,
forAll i,j : S. R((i,j)) = R((j,i))
```

Result:

```
letting R be
      function((1, 1) --> 1, (1, 2) --> 2, (1, 3) --> 3, (1, 4) --> 4,
               (2, 1) --> 2, (2, 2) --> 1, (2, 3) --> 4, (2, 4) --> 3,
               (3, 1) --> 3, (3, 2) --> 4, (3, 3) --> 1, (3, 4) --> 2,
               (4, 1) --> 4, (4, 2) --> 3, (4, 3) --> 2, (4, 4) --> 1)
letting e be 4
letting inv be function(1 --> 1, 2 --> 2, 3 --> 3, 4 --> 4)
```

This gives us the same first solution. In fact, there is the same number of solutions (16) to this problem as the previous one, proving that all groups of size 4 are abelian! In fact, the smallest non-abelian group is size 60, and that is beyond the size of problems we can find all solutions to with our current, simple model.

7.8 Handcrafting Instance Generators in Essence

Authors: Joan Espasa Arxer and Christopher Stone

In modelling it is common to create an abstract model that expects some input parameters (Also known as "instances") which are required to run and test the model. In this tutorial we demonstrate how to use ESSENCE to handcraft a generator of instances that can be used to produce input parameters for a specific model.

7.8.1 Instances for the Knapsack problem

Here is the model of the Knapsack Problem from (<link to other tutorial>) - knapsack.essence

```
given number_items : int
letting items be domain int (1..number_items)
given weight, gain : function (total) items --> int
given capacity : int
find picked : set of items
maximising sum i in picked . gain(i)
    such that (sum i in picked . weight(i)) <= capacity
```

This model has 4 different "given" statements :

- number_items: an integer for number of items
- weight: a functions that associates an integer(weight) to each item
- gain: a function that associates an integer(gain) to each item
- capacity: an integer that defines the capacity of the knapsack

The first parameter is fairly simple and we can even write this parameter with some value by hand by writing on a separate file (which we will call items.param):

```
letting number_items be 20
```

The remaining 3 parameters are more complex and labourious to be defined (too much work to be done by hand!) so we are going to write an ESSENCE specification that will create them for us. The fundamental starting step is writing find statements for each variable we wish to generate and ensure that the names of the variable (identifiers) are left unchanged. We can do so by creating a new file called generator.essence and write:

```
given number_items : int
letting items be domain int(1..number_items)
find weight: function (total) items --> int(1..1000)
find gain: function (total) items --> int(1..1000)
find capacity: int(1..5000)
```

Solving the above model (by running 'conjure solve generator.essence items.param' on the console) will create a set of parameters for our knapsack model. However, these instances are not interesting enough yet.

```
language Essence 1.3
letting capacity be 1
letting gain be
        function(1 --> 1, 2 --> 1, 3 --> 1, 4 --> 1, 5 --> 1, 6 --> 1, 7 --> 1, 8 -->␣
↪1, 9 --> 1, 10 --> 1, 11 --> 1,
                 12 --> 1, 13 --> 1, 14 --> 1, 15 --> 1, 16 --> 1, 17 --> 1, 18 --> 1,
↪ 19 --> 1, 20 --> 1)
letting weight be
```

(continues on next page)

(continued from previous page)

```
        function(1 --> 1, 2 --> 1, 3 --> 1, 4 --> 1, 5 --> 1, 6 --> 1, 7 --> 1, 8 -->␣
↪1, 9 --> 1, 10 --> 1, 11 --> 1,
                 12 --> 1, 13 --> 1, 14 --> 1, 15 --> 1, 16 --> 1, 17 --> 1, 18 --> 1,
↪ 19 --> 1, 20 --> 1)
```

We can make our instances more interesting by adding constraints into our generator's model. The first thing we notice is that all values assigned are identical, a bit TOO symmetrical for our taste. One simple solution to this issue is ensuring that all weights and gains assignments are associated with distinct values. This can be done by imposing injectivity as a property of the function.

```
find weight: function (total, injective) items --> int(1..1000)
find gain: function (total, injective) items --> int(1..1000)
```

And the results would be

```
language Essence 1.3
letting capacity be 1
letting gain be
        function(1 --> 1, 2 --> 2, 3 --> 3, 4 --> 4, 5 --> 5, 6 --> 6, 7 --> 7, 8 -->␣
↪8, 9 --> 9, 10 --> 10, 11 --> 11,
                 12 --> 12, 13 --> 13, 14 --> 14, 15 --> 15, 16 --> 16, 17 --> 17, 18␣
↪--> 18, 19 --> 19, 20 --> 20)
letting weight be
        function(1 --> 1, 2 --> 2, 3 --> 3, 4 --> 4, 5 --> 5, 6 --> 6, 7 --> 7, 8 -->␣
↪8, 9 --> 9, 10 --> 10, 11 --> 11,
                 12 --> 12, 13 --> 13, 14 --> 14, 15 --> 15, 16 --> 16, 17 --> 17, 18␣
↪--> 18, 19 --> 19, 20 --> 20)
```

This gives us a slighly more interesting parameters set but it is not there yet The specific order that appears in the results is solver dependent. The default solver used by conjure is Minion and we can use an optional flag to have the variables assigned in a random order. This can be done with this command:

```
conjure solve generator.essence items.param --solver-options=-randomiseorder
```

Alternatively one can use another solver that uses randomness by default

```
language Essence 1.3
letting capacity be 2841
letting gain be
        function(1 --> 858, 2 --> 653, 3 --> 673, 4 --> 365, 5 --> 389, 6 --> 783, 7 -
↪-> 566, 8 --> 664, 9 --> 387,
                 10 --> 576, 11 --> 864, 12 --> 741, 13 --> 102, 14 --> 735, 15 -->␣
↪276, 16 --> 41, 17 --> 132,
                 18 --> 974, 19 --> 293, 20 --> 381)
letting weight be
        function(1 --> 946, 2 --> 435, 3 --> 796, 4 --> 653, 5 --> 291, 6 --> 101, 7 -
↪-> 924, 8 --> 988, 9 --> 854,
                 10 --> 952, 11 --> 228, 12 --> 189, 13 --> 88, 14 --> 270, 15 -->␣
↪868, 16 --> 903, 17 --> 743,
                 18 --> 396, 19 --> 174, 20 --> 446)
```

Now it is starting to look more like a proper instance. At this point we can add some knowledge about the problem to formulate some constraints that will ensure that the instances are not trivial. ie when the sum of all the weights is smaller than the capacity so we can't put all the objects in the knapsack or when all the objects are heavier than the capacity so that no object can be picked. Thefore we add constraints such as:

```
such that (sum ([w | (_,w) <- weight]) > capacity*2)
```

This means that the sum of all the weights should be greater than twice the capacity of the knapsack. From this we can expect that on average no more than half of the objects will fit in the knapsack. The expression `[w | (_,w) <- weight]` is a list comprehension that extracts all right hand values of the `weight` function. The underscore character means we do not care about the left hand side values. To ensure that the solver does not take it too far we impose an upper bound using a similar constraint. We impose that the sum of the objects weights 5 times the capacity of the knapsack, so we can expect that only between 20% and 50% of the items will fit in the knapsack in each instance.

```
such that (sum ([w | (_,w) <- weight]) < capacity*5)
```

At this point it will be harder to see specific properties of the instances just by eyeballing the parameters but we can be confident that the properties we have imposed are there. We can add some more constraints to refine the values of the instances for practice/exercise by enforcing that no object is heavier than a third of the knapsack capacity

```
such that forAll (_,w) in weight .  w < capacity / 3
```

On top of that we can enfore a constraint on the density of the values in each object by limiting the ratio between the weight and gain of each specific object with:

```
such that forAll element : items .
        gain(element) <= 3*weight(element)
```

Finally the model of the generator is now :

```
given number_items : int
letting items be domain int(1..number_items)

find weight: function (total, injective) items --> int(1..1000)
find gain: function (total, injective) items --> int(1..1000)
find capacity: int(1..5000)
such that (sum ([w | (_,w) <- weight]) > capacity*2)
such that (sum ([w | (_,w) <- weight]) < capacity*3)
such that forAll (_,w) in weight .  w < capacity / 3
such that forAll element : items .
            gain(element) <= 3*weight(element)
```

After running once again the solver we can take the output solution file *generator-items.solution* and append it to the items.param (by concatenating the files or simply coping the content into it) We can finally test our instance by running "conjure solve knapsack.essence items.param"

Tada! your model is being tested on some instance!

If your computer is powerful enough you can try larger values in "letting number_items be 20" (40-50 items will already produce substantially harder instances) Like for other forms of modelling writing instance generators is in large part an art. If this is not your kind of thing and you would like a fully automated system that can produce instances you may check out this method [code available here]

7.9 Simple Permutations

Authors: Ruth Hoffmann and Gökberk Koçak, edited by András Salamon

7.9.1 Problem

Let a permutation be a sequence of length n consisting of numbers between 1 and n, in any order with each number occurring exactly once.

An interval in a permutation σ is a factor of contiguous values of σ such that their indices are consecutive. For example, in the permutation $\pi = 346978215$, $\pi(4)\pi(5)\pi(6) = 978$ is an interval, whereas $\pi(1)\pi(2)\pi(3)\pi(4) = 3469$ is not. It is easy to see that every permutation of length n has intervals of length 0, 1 and n , at least. The permutations of length n that **only** contain intervals of length 0, 1 and n are said to be simple. So for example the permutation $\pi = 346978215$ is not simple as we have seen in the example above that it contains an interval, on the other hand $\sigma = 526184937$ is simple as there are no intervals of length strictly greater than 1, except the whole of σ. See [Hof15] for more information on permutation patterns and simple permutations.

7.9.2 Enumeration/Generation Model

```
language Essence 1.3
given n : int
find perm : sequence (bijective, size n) of int(1..n)
such that
    and([ max(subs) - min(subs) + 1 != |subs| |
        i : int(1..n-1), j : int(2..n),
        i < j,
        !(i = 1 /\ j = n),
        letting subs be [perm(k) | k : int(i..j)]]
    )
```

```
given n : int
find perm : sequence (bijective, size n) of int(1..n)
```

We define the size of the permutation to be `n` and we are trying to find all the permutations `perm` to contain the integers 1 to `n`, by specifying that it is `bijective`.

```
```

The idea of our approach is the property of an interval, where when sorted it creates a complete range. This can be translated to checking that the difference between the maximal and the minimal elements of the interval is not equal to the cardinality of the interval.

We have one constraint to say that there are only intervals of length 0,1 and `n`. This constraint is defined as a matrix comprehension, which will build a matrix consisting of only boolean entries. We then check the matrix with an `and` constraint, to spot if there are any `false` entries, which would mean that we have an interval.

```
[ num | num : int(1..5), num != 3 ]
```

This is an example which creates a 1-dimensional matrix of `num` s where none of the entries are `3`. We allow also for `letting` statements inside the matrix comprehensions, which allow us to define intermediary statements.

```
and([ max(subs) - min(subs) + 1 != |subs| |
    i : int(1..n-1), j : int(2..n),
    i < j,
    !(i = 1 /\ j = n),
    letting subs be [perm(k) | k : int(i..j)]]
)
```

We extract `i` and `j` to be the beginning and the end of the interval, and we need to make sure that `i` is less than `j` to have the right order. As we do not want to include the whole permutation as an interval, we restrict that `i` and `j` cannot be simultaneously at the respective ends of the permutation. The final line of the comprehension sets up the continuous subsequences. On the left hand side of the matrix comprehension we use the interval property that when it is turned into a sorted set it is a complete range.

7.9.3 Parameter

```
language Essence 1.3
letting n be 5
```

7.9.4 Solving

Using the ESSENCE pipeline, we can solve our sample size by typing the following:

```
conjure solve simple_perm-gen-model.essence simple_perm-gen-param.essence-param
```

You will be then returned one `.solution` file, which contains a sample solution such as:

```
language Essence 1.3

letting perm be sequence(2, 4, 1, 5, 3)
```

If you look for an enumeration (or generation) of all solutions, type:

```
conjure solve simple_perm-gen-model.essence simple_perm-gen-param.essence-param --
↪number-of-solutions=all
```

The results will be saved into many (for `n be 5` you should get 6) `.solution` files which will contain a solution each.

7.9.5 Checking Model

```
language Essence 1.3
given n : int
given perm : sequence (bijective, size n) of int
find result : bool
such that
    result = and([ max(subs) - min(subs) + 1 != |subs| |
        i : int(1..n-1), j : int(2..n),
        i < j,
        !(i = 1 /\ j = n),
        letting subs be [perm(k) | k : int(i..j)]]
    )
```

```
given n : int
given perm : sequence (size n) of int
```

We define the size of the permutation to be `n` and the permutation `perm` to contain integers.

```
find result : bool
```

What the model will tell us is that the permutation is simple (`true`) or not.

7.9.6 Instances

```
letting n be 5
letting perm be sequence( 1, 4, 2, 5, 3)
```

This a non-simple permutation.

```
letting n be 5
letting perm be sequence(2, 4, 1, 5, 3)
```

This is a simple permutation.

7.9.7 Solving

Using the ESSENCE pipeline, we can solve our sample instance by typing the following:

```
conjure solve simple_perm-check-model.essence simple_perm-check-instance.essence-param
```

The result will be saved into a `.solution` file which will look something like this (or say `false`):

```
language Essence 1.3

letting result be true
```

CHAPTER

EIGHT

CONTACT

Conjure's main developer is Özgür Akgün. Please get in touch via email if you have comments, suggestions, or if you encounter any problems.

You can also use the issue tracker to report bugs.

We are particularly interested in hearing specific comments about the documentation. Please let us know if something is hard to understand, not easy to follow, or if the documentation is too sparse at a certain place. We will do our best to help!

8.1 Contributors

The following list of people have contributed to the development of Conjure.

- Özgür Akgün
- Alan Frisch
- Ian Gent
- Brahim Hnich
- Bilal Syed Hussain
- Chris Jefferson
- Ian Miguel
- Peter Nightingale

BIBLIOGRAPHY

[AkgunDM+19] Özgür Akgün, Nguyen Dang, Ian Miguel, András Z. Salamon, and Christopher Stone. Instance generation via generator instances. In Thomas Schiex and Simon de Givry, editors, *CP 2019: Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming*, volume 11802 of Lecture Notes in Computer Science, 3–19. Springer, 2019. doi:10.1007/978-3-030-30048-7_1.

[CDDC+19] Sara Ceschia, Nguyen Dang, Patrick De Causmaecker, Stefaan Haspeslagh, and Andrea Schaerf. The second international nurse rostering competition. *Annals of Operations Research*, 274:171–186, 2019. doi:10.1007/s10479-018-2816-0.

[Dud24] H. E. Dudeney. Perplexities. *Strand Magazine*, 68:94,214, July 1924.

[FHJ+08] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: a constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. doi:10.1007/s10601-008-9047-y.

[Hof15] Ruth Hoffmann. *On dots in boxes, or Permutation pattern classes and regular languages*. PhD thesis, University of St Andrews, 2015. URL: https://research-repository.st-andrews.ac.uk/handle/10023/7034.