Conjure Documentation

Release 2.1.0

Özgür Akgün

Apr 08, 2018

Table of Contents

CHAPTER 1

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 2

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 3

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, MacOS, and Windows. 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.

3.2 Compiling from source

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

```
git clone git@github.com:conjure-cp/conjure.git
cd conjure
make
```

Installation is known to work with GHC-7.10.3, GHC-8.0.2, and GHC-8.2.2, and

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.

CHAPTER 4

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.

CHAPTER 5

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 6

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 two variants:

1. First variant takes two arguments where the first argument is the syntax of the items of the list and second argument is the item separator.
2. Second variant takes an additional third argument which indicates the surrounding bracket for the list. The third argument can be one of round brackets (`()`), curly brackets (`{}`), or square brackets (`[]`).

```
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.

```
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
        | "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 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.

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.

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.

To explicitly specify a matrix, use a list of values inside square brackets.

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

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)
```

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. 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.

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`, `subsequence(s,t)` tests whether there is a function `f` such that the list of values taken by `s` occurs in the same order in the list of values taken by `t`, and `substring(s,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`.

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
```

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
```

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. However, avoid using the same name both for quantification and as a global decision variable in a `find`, as this is treated as an error by Savile Row.

An alternative quantifier-like syntax

```
sum i in I . 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 the usual square brackets `[` and `]`, inside which is a generator expression possibly involving some parameter variables, followed by `|`, followed by a comma (`,`) separated sequence of conditions defining the values that all the parameter variables may take, or Boolean expressions. 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 Boolean expressions evaluate to `true`.

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.

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
```

CHAPTER 7

Demonstrations

We demonstrate the use of Conjure for some small problems.

7.1 Number puzzle

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. We then need to express what it means for the digits of the sum to behave as we expect; a natural approach is to introduce carry digits and use those to express the sum digit-wise. There are only at most two digits and a carry digit being added at each step, so the carry digits cannot be larger than 2.

```
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)
find carry1,carry2,carry3,carry4 : int(0..2)
such that
           f(D) + f(E) = f(Y) + 10*carry1,
  carry1 + f(N) + f(R) = f(E) + 10*carry2,
  carry2 + f(E) + f(O) = f(N) + 10*carry3,
```

```
  carry3 + f(S) + f(M) = f(O) + 10*carry4,
  carry4 = f(M)
```

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 to avoid repetition.

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 carry1 be 0
letting carry2 be 0
letting carry3 be 0
letting carry4 be 0
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 carry1 be 1
letting carry2 be 0
letting carry3 be 1
letting carry4 be 0
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)
find carry1,carry2,carry3,carry4 : int(0..2)
such that
           f(D) + f(E) = f(Y) + 10*carry1,
```

```
  carry1 + f(N) + f(R) = f(E) + 10*carry2,
  carry2 + f(E) + f(O) = f(N) + 10*carry3,
  carry3 + f(S) + f(M) = f(O) + 10*carry4,
  carry4 = f(M),
  M > 0, S > 0
```

Let's try again:

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

This now leads to the solution we expected:

```
letting carry1 be 1
letting carry2 be 1
letting carry3 be 0
letting carry4 be 1
letting f be function(S --> 9, E --> 5, N --> 6, D --> 7, M --> 1,
  O --> 0, R --> 8, Y --> 2)
```

Note that the solution includes both the mapping we were looking for, as well as values for the carry digits that were introduced to express the constraints.

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 Labelled connected graphs

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.

7.2.1 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
$ 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.

7.2.2 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)];
   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.

7.2.3 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.

7.2.4 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.

7.2.5 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.

7.2.6 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.

CHAPTER 8

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

[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.