Conjure Documentation

Release 2.0.0

Özgür Akgün

Jul 10, 2017

Contents

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 1

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!

CHAPTER 2

Table of Contents

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.

Installation

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

Downloading a binary

Conjure is available as a binary for most platforms. 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.

TODO: link to executables will come here.

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 the commonly available cabal tool.

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

It is known to work with GHC-7.8.4 and GHC-7.10.3.

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.

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 flags. A flag may or may not take a value. 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 flag does not require a value. However, the `conjure modelling` command takes a flag called `--output-directory`, which specifies the directory under which Conjure places its output files. This flag requires a value.

Flags 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` a long name for a flag, and `-o` is a short name for the same flag.

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.

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`.

Help output

Following is Conjure's full help message, provided for reference.

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*.

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. In order 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.

High level of abstraction

Conjure's input language is Essence. Essence provides abstract domain types like matrices, sets, multi-sets, functions, sequences, relations, partitions, records, and variants. These abstract domain types also support domain attributes like cardinality for set-like domains, injectivity/surjectivity for functions, and etc to enable concise specification of a problem. Essence also provides more primitive domain types, like booleans, integers and enumerated types, 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*.

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.

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. In order 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.

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 very often a problem can be modelled in several ways, and it is very hard to know what a *good* model is for a given problem. Constraint programming experts spend considerable amounts of time developing models. Often, in order to choose a good model, multiple models are created only to evaluate their relative performances for solving the problem at hand.

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 to solve.

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.

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.

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 very small number.

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

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.

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 is *[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 (`[]`).

Problem Specification

```
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.

A problem specification can contain at most one objective statement.

Statements

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

There are five kinds of statements in Essence.

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 to an expression or a domain (`LettingStatement`), and enumerated or unnamed types. The syntax for each of these declaration statements are given in the following.

Declaring decision variables

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

A decision variable is declared by using the "find" keyword, 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.

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 "given" keyword instead of the "find" keyword. Unlike decision variables, the domains of parameters do not have to be finite.

Declaring aliases

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

An alias to an expression can be declared by using the "letting" keyword, followed by the name of the alias, followed by the keyword "be", followed by an expression. Similarly, an alias to 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 to `y + z` and `d` is declared as a domain alias to `set of int(a..b)`.

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 only support equality and ordering operators; they do not support arithmetic operators.

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.

Search directives

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

BranchingOn := Name
             | Expression
```

Essence is a high level problem specification language and typically it doesn't include lower level details such as search directives. In fact the reference paper on Essence (*[FHJ+08]*) does not include these search directives at all.

For pragmatic reasons we support adding 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-on statement.

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.

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 the later sections.

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

An objective can be declared by using either the "minimising" or the "maximising" keyword 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.

Names

The lexical rules for valid names in Essence are very similar to those of most common languages. A name consists of a sequence of non-whitespace alphanumeric characters (letters or digits). The first character of a valid name has to be a letter or an underscore character ('_').

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.

Boolean domains

A Boolean domain is denoted with the keyword "bool", and has two values: "false" and "true".

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.

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.

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.

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.

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.

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.

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.

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”.

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.

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.

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.

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.

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 N 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.

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 rules 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.

Expressions

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

Operator := ...
```

(In preparation)

Matrix indexing

Tuple indexing

Demonstrations

(In preperation)

References

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!

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

[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.