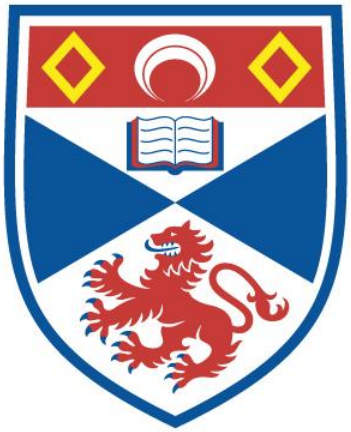
EXTENSIBLE AUTOMATED CONSTRAINT MODELLING VIA REFINEMENT OF ABSTRACT PROBLEM SPECIFICATIONS

ÖZGÜR AKGÜN

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2014

Extensible Automated Constraint Modelling via Refinement of Abstract Problem Specifications

Özgür Akgün

May 2014

Extensible Automated Constraint Modelling via Refinement of Abstract Problem Specifications

THESIS

submitted to the

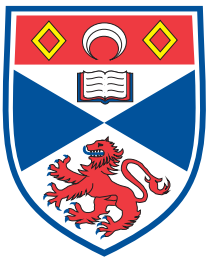
University of St Andrews

for the degree of

Doctor of Philosophy

by

Özgür Akgün



University
of
St Andrews

May 2014

Abstract

Constraint Programming (CP) is a powerful technique for solving large-scale combinatorial (optimisation) problems. Constraint solving a given problem proceeds in two phases: *modelling* and *solving*. Effective modelling has an huge impact on the performance of the solving process. This thesis presents a framework in which the users are not required to make modelling decisions, concrete CP models are automatically generated from a high level problem specification. In this framework, modelling decisions are encoded as generic rewrite rules applicable to many different problems.

First, modelling decisions are divided into two broad categories. This categorisation guides the automation of each kind of modelling decision and also leads us to the architecture of the automated modelling tool.

Second, a domain-specific declarative rewrite rule language is introduced. Thanks to the rule language, automated modelling transformations and the core system are decoupled. The rule language greatly increases the extensibility and maintainability of the rewrite rules database. The database of rules represents the modelling knowledge acquired after analysis of expert models. This database must be easily extensible to best benefit from the active research on constraint modelling.

Third, the automated modelling system Conjure is implemented as a realisation of these ideas; having an implementation enables empirical testing of the quality of generated models. The ease with which rewrite rules can be encoded to produce good models is shown. Furthermore, thanks to the generality of the system, one needs to add a very small number of rules to encode many transformations.

Finally, the work is evaluated by comparing the generated models to expert models found in the literature for a wide variety of benchmark problems. This evaluation confirms the hypothesis that expert models can be automatically generated starting from high level problem specifications. An method of automatically identifying good models is also presented.

In summary, this thesis presents a framework to enable the automatic generation of efficient constraint models from problem specifications. It provides a pleasant environment for both problem owners and modelling experts. Problem owners are presented with a fully automated constraint solution process, once they have a precise description of their problem. Modelling experts can now encode their precious modelling expertise as rewrite rules instead of merely modelling a single problem; resulting in reusable constraint modelling knowledge.

Candidate's declaration

I, Özgür Akgün, hereby certify that this thesis, which is approximately 32000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in September 2009 and as a candidate for the degree of Doctor of Philosophy in September 2009; the higher study for which this is a record was carried out in the University of St Andrews between 2009 and 2013.

date _____ *signature of candidate* _____

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date _____ *signature of supervisor* _____

Permission for electronic publication

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. We have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by the candidate and the supervisor regarding the electronic publication of this thesis:

Access to Printed copy and electronic publication of thesis through the University of St Andrews.

date _____ *signature of candidate* _____

date _____ *signature of supervisor* _____

Acknowledgements

I would like to thank my supervisors Ian Miguel and Chris Jefferson for doing the best possible job in advising me throughout my PhD studies. They helped me vastly improve my skills and knowledge by providing constant support, encouragement, and friendship. Thank you for giving me a lot of freedom to explore my own ideas yet always being there for the very interesting discussions.

I am thankful to two people I met during my undergraduate studies who are probably responsible for my decision to even consider doing a PhD in the first place. I thank Armagan Tarim for being a wonderful lecturer during the two Operations Research courses I took from him and for introducing me to Brahim Hnich, with whom I always enjoyed working together outside of lectures. They were fantastic role models in the early days of my education.

Working with all the members of our Constraints group has been a great experience for me. I am thankful to Ian Gent, Peter Nightingale, Andrea Rendl, Lars Kotthoff, Neil Moore, Arunas Prokopas, Bilal Hussain, and James Wetter. I also want to thank researchers outside of St Andrews, Alan Frisch and Karen Petrie.

Many people have helped with their friendship during my time in St Andrews. I am sure I will fail to write a complete list but, amongst many others, I want to thank Blesson Varghese, Chris Choi, CJ Davies, David Letham, Edwin Brady Jamie Carson, Jan de Muijnck-Hughes, Jaunty Ward, Lakshitha De Silva, Luke Hutton, Masih Hajiarabderkani, Ruth

Hoffmann, Saray Shai, Stuart Norcross, Tom Kelsey, and Ward Jaradat. I am also thankful to the large number of friends I have outside of St Andrews; thanks for staying so closely in touch despite the long distances.

My whole family has always been hugely supportive and encouraging to me. I especially thank my brother Utku, mother Birgül and father Musa for being the closest to me.

I thank both my external examiner Steven Prestwich and internal examiner Kevin Hammond for providing a large number of helpful corrections, and tolerating a submitted thesis which was less polished than I would have liked. The final version of the thesis has improved greatly thanks to your feedback and obviously any remaining mistakes are my own!

This thesis is dedicated to Emine. Without your love and constant support, none of this would have been possible. I love you with all my heart.

Contents

Chapter 1

Introduction

Constraint Programming (CP) can be a very powerful way for solving difficult combinatorial problems; however, using CP *effectively* requires a lot of expertise and hard work, even for experts. Solving a problem using CP proceeds in two steps: *modelling* and *solving*. Problems are often communicated in informal ways. On the other hand, CP solvers are computer programs and they need formal descriptions of problems: CP models. Modelling is the process of producing a concrete CP model for a given problem. It requires several modelling decisions to be made, since typically there are many ways to model the same problem. A CP model contains decision variables with associated domains and constraints which limit assignments to the decision variables. Solving a CP model is the process of finding a simultaneous assignment to all decision variables without violating any constraints.

The importance of modelling in CP is acknowledged by Barbara Smith in Chapter 11 of [Smi06a]: *"... there is abundant evidence that how the problem to be solved is modelled as a Constraint Satisfaction Problem (CSP) can have a dramatic effect on how easy it is to find a solution, or indeed whether it can realistically be solved at all."* Different models for the same problem, despite all being correct, can take drastically different amounts of time to solve using the same solver. Unfortunately, it is very hard to compare models for effectiveness without actually trying each alternative. Hence, even expert modellers often need to experiment with different ways of modelling a problem until they reach a good model. Moreover, typically a

modelling technique which works very well for a certain problem may not work so well for another problem.

The difficulty of formulating an effective model is also referred to as the *modelling bottleneck* and is considered to be one of the key challenges facing the constraints field [Pug04], and one of the principal obstacles preventing widespread adoption of constraint solving. In order to address this challenge, automated modelling has become a very active area of research in CP. Several approaches have been taken to automate aspects of CP modelling including machine learning [Bes+06]; case-based reasoning [Lit+03]; theorem proving [Cha+06]; automated transformation of medium-level solver-independent constraint models [Ren10; Net+07; Van99; Mil+99]; and refinement of abstract constraint specifications [Fri+05b; Akg+11b] in languages such as ESRA [Fle+03], Essence [Fri+08], $\mathcal{F}$ [Hni03] or Zinc [Mar+08a; Kon+10].

The approach described in this thesis is to start from a highly abstract *problem specification* and produce concrete CP models automatically. The problem specification language Essence enables specifying problems concisely; the language is designed to capture problem structure at a level of abstraction that is above where most CP modelling decisions are made. For instance, having an array of decision variables and posting an `allDiff` constraint on this array is a common idiom in CP modelling. Many uses of this idiom can be replaced with a set decision variable. Using a set decision variable instead, the user gets access to a large collection of predefined set operators — such as union, intersection, subset, set equality — and more importantly they do not commit to using a one dimensional array. Conjure, the automated constraint modelling tool developed as a part of this thesis, can *refine* this decision variable and all expressions involving it in multiple ways, possibly also including the one-dimensional array representation as an alternative.

In addition to sets, Essence provides decision variables with other abstract domains: tuples, enumerations, functions, relations, multi-sets, partitions, and allows arbitrary nesting of these. It also provides a rich collection of operators for variables with abstract domains enabling concise specification of problem structure. Conjure applies modelling transform-

ations to the input problem specification to produce CP models. In order to express the transformation rules, Conjure uses a domain-specific language. Using a domain-specific language enables us to write rules more easily, extend Conjure's modelling capabilities without recompiling it, and make rule authoring more accessible to CP researchers so they can encode new modelling *tricks* and improve Conjure.

Conjure contains at least one, but typically several, representation options for each abstract domain, and alternative translations for operators on abstract variables. Hence, it can typically generate several alternative formulations of a single problem specification. In this regard, Conjure is similar to conventional compilers, which also have to choose between alternative transformations during compilation. Each decision in Conjure is far more important though, since solution performance of different models for the same problem can be drastically different.

1.1 Background

Conjure was introduced in prototype form by Frisch *et al.* [Fri+05b]. It was able to refine a fragment of Essence limited to nested set and multi-set decision variables into models in Essence′, a solver-independent modelling language. Subsequent work [Mar+06; Mar+08b], considered issues involved in automatically channelling among different representations of abstract variables. These were research prototypes which should be considered as experimental. They were only able to refine a fraction of the input language Essence. The Conjure system presented in this thesis is implemented from scratch and is a major step forward over the previously reported prototypes. In particular it is able to refine Essence specifications using all abstract types and operators of Essence, and it is shown to produce effective models for several benchmark problems in CP.

Essence [Fri+08] is a language for specifying combinatorial (decision or optimisation) problems. It has a high level of abstraction to allow users to specify problems *without* making constraint modelling decisions, supporting decision variables whose types match

```
1  language Essence 1.3
2
3  given w, g, s : int(1..)
4  letting Golfers be new type of size g * s
5  find sched: set (size w) of
6                 partition (regular, size g) from Golfers
7
8  such that
9    forAll week1, week2 in ached, week1 != week2 .
10     forAll group1 in parts(week1) .
11       forAll group2 in parts(week2) .
12         |group1 intersect group2| < 2
```

Figure 1.1: Essence specification of the Social Golfers Problem

the combinatorial objects problems typically ask us to find, such as: sets, multi-sets, functions, relations and partitions. First introduced by the language is its support for the *nesting* of these types, allowing decision variables of type set of sets, multi-set of sets of functions, *etc*. Hence, problems such as the Social Golfers Problem [Har01], which is naturally conceived of as finding a set of partitions of golfers subject to some constraints, can be specified directly without the need to model the sets or partitions as matrices.

An Essence specification (see [Fri+08] for full details), such as that in Figure 1.1, identifies: the parameters of the problem class (`given`), whose values are input to specify the instance of the class; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy to be a solution (`such that`). An objective function may also be specified (`min/maximising`) and, for concision, identifiers may be declared (`letting`).

Today's constraint solvers typically support decision variables with atomic types, such as integer or Boolean, have limited support for more complex types like sets or multi-sets, and no support for nested complex types. Hence, abstract specifications are *refined* by modelling abstract decision variables as constrained collections of variables of unnested primitive types. The system developed as a part of this thesis, unlike the older prototypes, employs a system of rules to refine Essence specifications into constraint models in Essence′ [Ren10], a language derived from Essence mainly by removing facilities for abstraction and adding

facilities common to existing constraint solvers and toolkits. From Essence′ a tool such as Tailor [Ren10] can be used to translate the model into the format required for a particular constraint solver.

An abstract specification typically can be implemented by many alternative concrete constraint models. Conjure is intended to generate these alternatives by providing multiple refinement rules for each abstract type, corresponding to the various ways in which a decision variable of that type can be modelled. Furthermore, for each way of modelling the decision variables there can be multiple rules to generate alternative models for a constraint on those variables. Consequently, Conjure often generates many alternative models for an input specification. We aim to encode each rule that for some problem is used in the generation of some good (or perhaps reasonable) model. Given a problem specification and a set of rules the system generates all possible models. If we have encoded a sufficient set of rules, then the kernels of all good (or reasonable) models of the problem should be contained within the set of models. An alternative mode of operation would be to try to restrict the set of models produced by Conjure to contain only the *good* models, however this is not desirable at this point because 1) it is a much harder problem to only produce the good models 2) models may have complementary strengths and producing a diverse portfolio of models can be very valuable as well.

This thesis focuses on the refinement-based approach, in which a user writes abstract constraint specifications to describe a problem at a higher level than that where modelling decisions are normally made. Abstract constraint specification languages, e.g. Essence and Zinc, support abstract variables with types for common mathematical structures such as sets, multi-sets, functions, and relations, as well as nested types, such as set of sets and multi-set of functions. Problems can often be specified very concisely in this way. For example, the Social Golfers Problem [Har01], which is naturally conceived of as finding a set of partitions of golfers subject to some constraints, can be specified directly (see Figure 1.1) without the need to model the sets or partitions as matrices.

An Essence specification, such as that in Figure 1.1, identifies: the input parameters

of the problem class (`given`), whose values define an instance; the combinatorial objects to be found (`find`); and the constraints the objects must satisfy (`such that`). An objective function may also be specified (`min/maximising`) and identifiers may be declared (`letting`).

Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. The Conjure system[Akg+11b] uses refinement rules to convert an Essence specification into the solver-independent constraint modelling language Essence′ [Ren10]. From Essence′ we use SavileRow[1] to translate the model into input for a particular constraint solver while performing solver-specific model optimisations.

1.2 Publications

Research described in this thesis appeared partly in previous papers of mine. Most of these papers were co-authored with Ian Miguel and Chris Jefferson, my two PhD supervisors. Other co-authors in these papers are Alan Frisch, Brahim Hnich, Ian Gent, Peter Nightingale, Lars Kotthoff, and Bilal Hussain.

All of the publications listed here are based primarily on my work.

[Akg+10b; Akg+10c; Akg+10a] were published in the first year of my PhD. They should be considered as preliminary publications, mainly presenting the idea of generating multiple alternative CP models starting from a high level problem specification language.

[Akg+11b] is the first publication which describes a concrete implementation of Conjure and provides extensive evaluation of its capabilities. It describes how Conjure works and proves the hypothesis that kernels of published CP models can be produced by an automated modelling tool.

[Akg+11a] can be considered as an application paper. The Open Stacks problem was the problem used for the Constraint Modelling Challenge in 2005. This paper presents a 6-years too late entry to the challenge. Conjure initially produced correct but poor models for this problem; however, as a result of this exercise some new modelling transformations

[1] `http://savilerow.cs.st-andrews.ac.uk`

were formulated to improve the generated models. These modelling transformations were recorded in order to potentially improve Conjure's modelling capabilities for other problems as well.

[Akg+13b] presents two main contributions. First is generating symmetry breaking constraints automatically, which improves models generated by Conjure beyond model kernels. Second is two methods for automated model selection: racing and the Compact heuristic.

[Akg+13a] presents the complete chain of tools we work on in the Constraints group at the School of Computer Science in University of St Andrews. In addition to Conjure, SavileRow, Minion and Dominion are in this tool-chain.

[Akg+13c] is a workshop paper describing current achievements and future directions in automated modelling and automated model selection in CP.

1.3 Contributions

The main contribution of this thesis is the demonstration of the refinement based approach to automated modelling in CP. Achieving this ambitious goal requires several techniques such as proper handling of nested domains for decision variables, automatically posting structural constraints for decision variables with abstract domains, and refining constraint expressions with respect to the refinement of domains. Symmetry breaking constraints and channelling constraints need to be posted automatically. Moreover, in order to increase the utility of automated modelling, automated model selection is essential. Finally, modelling transformations need to be encoded in a language as close as possible to CP modelling languages in order to facilitate ease of use.

Following is a list of contributions of this thesis, each individually elaborated throughout the thesis and all collectively supporting the main contribution of the thesis: achieving effective automated modelling in CP using a refinement based approach.

Refinement of arbitrarily nested domains In Essence, abstract domain constructors can

be nested arbitrarily to create the domains of decision variables and parameters. The techniques presented in this thesis are able to handle the refinement of domains with arbitrary levels of nesting.

Symmetry breaking Demonstrating how modelling symmetry can be broken as soon as it enters the model to improve the produced models beyond model kernels.

Automated channelling Ability to represent a single abstract decision variable in multiple ways and post channelling constraints between different representations automatically.

Finer grained expression refinement In comparison to the prototype implementations of Conjure, the work presented in this thesis avoids flattening of expressions. This avoids generation of unnecessary auxiliary variables. It also greatly reduces the number of required rules to achieve full coverage of Essence.

Conjure As a part of this thesis the automated constraint modelling tool Conjure is developed from scratch. This allowed me to empirically test ideas about automated model generations.

Full coverage of Essence There was a prototype implementation of Conjure partly presented in [Mar+08b]. However that implementation was very limited, it did not support all of Essence. It was particularly a prototype to study automated generation of channelling constraints.

Rule language A domain-specific rule language is designed and implemented to encode modelling transformations. This enables easier authoring and maintenance of rules.

Extensibility Thanks to the rule language, finer grained expression refinement, and representation independent (horizontal) rules extending Conjure with new variable representations is very easy. This was a particularly desired feature because new ways of modelling existing problems is discovered continuously.

Racing A model selection technique which can be used when a representative collection of instance data is available for a problem class.

The Compact heuristic A light-weight model selection heuristic which tries to generate the most *compact* model for a problem class. This is considered light-weight because it does not need any instance data and produces a single model without any further analysis.

Evaluation Conjure's ability to successfully refine real-life problem specifications and produce kernels of published models is experimentally evaluated. The model selection techniques are also experimentally evaluated.

1.4 Thesis structure

This thesis is structured as follows.

Chapter 2 outlines related work in the area of automated modelling in CP and program transformation in general. Chapter 3 gives a complete example problem specification in Essence, followed by a step by step description of Conjure's operation on this problem specification. This chapter is aimed to give an intuitive understanding of the general ideas presented in the thesis. Chapter 4 describes the design and architecture of the automated modelling tool Conjure. It does not go into details about how modelling transformations are defined, instead it describes how the transformations are applied.

Chapter 5 describes the rule language together with its features and operators. It also describes the three different kinds of rules: representation decision rules, horizontal expression refinement rules, and vertical expression refinement rules. Chapter 6 gives a listing of actual rules that are in Conjure's rules database. The chapter is structured to contain a section for each abstract type constructor in Essence. Each section contains a number of subsections, one for each representation option. The subsections contain the corresponding representation selection rules and vertical rules specific to the representation.

In addition to these sections, horizontal rules are given in a section of their own since they are representation independent.

Chapter 7 demonstrates how by only adding a few rules a new representation for sets, the Gent representation, can be added for set variables; and Remarkably, only two rules are needed to get a fully working representation. A third rule is added to demonstrate how new vertical rules can be added to improve the quality of generated models. A new representation for partial functions is also added using only a representation selection rule and two vertical expression refinement rules.

Chapter 8 describes automated symmetry breaking in Conjure. Adding symmetry breaking improves the quality of CP models produced by Conjure drastically. Chapter 9 demonstrates how Conjure can be applied to a wide range of problem specifications. This chapter also shows that some of the produced models are actually interesting: they were published in peer reviewed publications by CP experts. In addition, two model selection methods are described to evaluate Conjure's ability to identify effective models from among all models it can generate.

Chapter 10 concludes the thesis by giving a summary and a discussion of future research directions. The experimental results of this thesis can be found at `http://ozgur.host.cs.st-andrews.ac.uk/thesis/experiments.zip`.

Chapter 2

Related Work

This chapter describes the related work in the areas of modelling in CP, automated modelling in particular, and approaches to program transformation in general.

2.1 Refinement based approaches

A very popular approach to automated modelling in CP is using a high-level problem specification language together with automated refinement of concrete CP models. Automated refinement is the process of successively translating the problem specification into a concrete CP model, where the model expresses the decision variables and constraints explicitly. Concrete CP models are then solved using a constraint solver and solutions are translated back to the high-level language. The most important tools and languages which took this approach are presented in the rest of this section.

2.1.1 OPL

OPL [Van+99; Van99] is a modelling language for mathematical programming and constraint programming. It is generally regarded as the first high level modelling language; before OPL the intercase to CP solvers was through directly manipulating internal data structures of a solver. OPL offers decision variables with integer and enumerated variables, and only operators relating to these types of variables. OPL does not offer abstract domains.

2.1.2 ESRA

ESRA [Fle+03] is a language with special focus to decision variables with relation domains. It is translated to OPL by refining relation domains and operators. It does not however consider multiple alternative ways when doing refinement; each abstract domain and constraint can be refined in only one way. Moreover, the abstract domains offered by ESRA cannot be nested arbitrarily.

2.1.3 NP-Spec

NP-Spec [Cad+00] is a language which allows the specification of NP-complete problems in a subset of existential second order logic. It provides a small number of high level domains –sets and partitions of integers– which are refined down to decision variables with simpler domains. Like ESRA, NP-Spec provides only one way to model each high-level domain and operator; hence, does not allow for the generation of alternative models.

2.1.4 F and Fiona

F [Hni03] is a language with function variables. Problems modelled in F are refined into OPL using a system called Fiona. F supports function attributes like total and bijective. Function domains in F cannot be nested arbitrarily, a function variable is simply a mapping between non-nested domains like integers or enumerations. In contrast to OPL, ESRA and NP-Spec, F considers multiple refinements of functions. It contains a number of heuristics to choose amongst different refinement options for function variables. Fiona always generates a single output model using these heuristics. If the same function variable is refined in multiple ways within a single model, Fiona is able to generate channelling constraints automatically.

2.1.5 CGRASS

The CGRASS [Fri+02] system explores the idea of reformulating CP models using a collection of rules in order to improve them. It does not change representations of decision variables;

however it can rearrange constraint expressions and reduce domains of decision variables. CGRASS is mostly problem instance based. The authors demonstrate how some implied constraints and symmetry breaking constraints can be added using a system of reformulation rules. CGRASS is limited to integer variables, and arithmetic and logical operators on integer expressions.

2.1.6 TAILOR and SAVILEROW

TAILOR [Ren10] is an automated modelling tool for the solver independent CP modelling language ESSENCE′. It can target multiple solvers and performs a variety of problem instance level reformulations, such as Common Sub-expression Elimination (CSE) and elimination of duplicate constraints. Recently TAILOR was renamed to SAVILEROW[1].

2.1.7 MiniZinc

MiniZinc [Net+07] is a medium-level constraint modelling language. It contains features common to many CP modelling languages such as boolean and integer domains, and arrays for collections of these variables. MiniZinc can be used to describe problem class models, however it does not perform and reformulations at the class level. When presented with problem instance data, the class model is instantiated into an instance model which can be targeted to one of several solver backends. MiniZinc uses a solver-dependent instance level language called FlatZinc to interact with solvers.

2.1.8 Zinc

Zinc [Mar+08a] is a higher level constraint modelling language in comparison to MiniZinc. It provides decision variables with set domains as well as user defined record-like domains. Zinc is compiled to MiniZinc, and in principle it can permit an exploration of different modelling choices. However, to the best of our knowledge, the existing Zinc compiler only produces one MiniZinc output model for an input high level model in Zinc.

[1] http://savilerow.cs.st-andrews.ac.uk

2.2 Example driven automated modelling

The tools and languages presented in this section take a very different approach to automated modelling in CP in comparison to the tools and languages presented in the previous section.

2.2.1 O'CASEY

O'Casey [Lit+03] is a case based reasoning tool. It uses recordings of previous problem solving episodes. Problems are paired with problem instances to form a *case*. The experience obtained from cases are mainly the selection of propagators and search heuristics. Hence, the reformulations provided by O'Casey do not change variable representations or the statement of constraint expressions.

2.2.2 Conacq

Conacq [Bes+06] is a SAT-based version space algorithm to acquire constraint networks. Its inputs are the set of decision variables, and a collection of positive and negative examples. Positive examples are valid solutions to the problem and negative examples and non-solutions. It automatically generates constraints by applying machine learning techniques.

2.2.3 Constraint and Model Seeker

The Constraint Seeker [Bel+11] and the Model Seeker [Bel+12] are examples of example-driven automated modelling in CP. In comparison to Conacq, these tools focus on the automated acquisition of global constraints. They use a large collection of positive and negative fully instantiated instance models to *learn* individual global constraints and complete models respectively.

2.3 Program transformation and refactoring

Program transformation is any operation that takes a computer program and generates another program. Refactoring is a special case of program transformation where generally

the external behaviour of the program is not effected by the transformations. In the case of automated modelling in CP we are focusing on a specific case of program transformation, however it is worth mentioning other applications of program transformation and reformulation since they are useful in a broader context.

In the simplest case, refactoring is used through Integrated Development Environments (IDEs) to reorganise program code. Most refactorings are trivial operations like renaming identifiers, adding new arguments to existing functions, or extracting function definitions from a selected code fragment.

Non-trivial uses of program transformation and refactoring include rearranging program code before or during compilation in order to generate highly-optimised executables.

As early as 1977, refactoring tools were used to transform recursive programs by eliminating unnecessary applications of *fold/unfold* functions [Bur+77]. More recently, the Haskell Refactorer (HaRe) was developed to provide users a large catalog of refactorings like renaming and lambda lifting [Bro+11]. There are small structural changes, however using multiple steps of such small changes users can perform non-trivial refactorings. The Paraphrase project [Bro+13; Ham+13] employs advanced refactoring techniques to restructure programs into a form which is more suited to parallel execution than the original program.

In addition to stand-alone program transformation tools, some modern compilers provide facilities to programmers to apply program transformations during compilation. For example, using rewrite rules in the Haskell compiler GHC programmers can express domain specific optimisations that the compiler can otherwise cannot discover by itself [Jon+01].

2.4 Summary

This chapter presents existing tools, languages and approaches for automated modelling in CP. Some of these approaches, similar to the approach presented in this paper, use program transformation techniques to translate models written in higher level languages down to

lower level languages that are closer to the input languages of CP solvers. Distinguishing features of Conjure and Essence are support for a rich collection of abstract domain constructors and arbitrarily nested types in the input language, operating at the problem class level instead of at the problem instance level, and the generation of multiple alternative models instead of a single model. Conjure also differs from existing tools by the use of a domain specific rewrite rule language and its special focus on ease of extensibility. Example driven approaches are also briefly surveyed. Instead of requiring a high level problem description from the user they only require a statement of decision variables and the constraint expressions are automatically *learned* from positive and negative examples of solutions for the original problem. Finally, the last section gives examples of program transformation and refactoring tools outside of the narrow context of automated modelling in CP.

Chapter 3

Conjure by Example

This chapter is a cross section of the most important contributions of this thesis. It demonstrates the operation of Conjure on a simple problem specification in Essence. In doing so, it exemplifies some constructs of the input language Essence and outlines the output language Essence′ in comparison. It also shows *what* transformations are applied to the given problem step by step to reach the final output of alternative CP models. However, it does not show *how* the transformations are encoded or implemented, those are covered in later chapters.

3.1 Essence and Essence′

Essence is a highly abstract problem specification language. Specifying problems in Essence does not require CP modelling decisions to be made. In order to facilitate a high level of abstraction, it provides decision variables with *abstract* domains and a rich set of operators defined on such decision variables.

On the other hand, Essence′ is a typical CP modelling language. It is only provides decision variables with *concrete* domains and operators defined on such decision variables.

A thorough definition of both languages is given in later chapters, only a brief discussion will be given in this section.

```
1  language Essence 1.3
2
3  given object new type enum,
4  given weight, value : function (total) object --> int(1..),
5  given maxWeight, minValue : int(1..)
6  given knapsackSize : int(1..)
7
8  find knapsack : set (size knapsackSize) of object
9
10 such that
11     maxWeight >= sum i in knapsack . weight(i),
12     minValue  <= sum i in knapsack . value(i)
```

Figure 3.1: An Essence problem specification of the Knapsack Problem

Concrete decision variable A concrete decision variable is one whose domain is directly supported in the target solver. Typically solvers support Booleans and Integers with finite domains. Most solvers also support arrays of decision variables with these domains. Essence′ is no exception, it supports `bool`, `int` and `matrix` domains for decision variables.

Abstract decision variable An abstract decision variable is one whose domain is not directly supported in the target solver. Such domains have to be *represented* using a collection of concrete decision variables. This usually requires posting additional constraints to maintain invariants of the original abstract domain. Essence supports multiple kinds of abstract domains, built using the domain constructors `set`, `mset`, `function`, `relation`, and `partition`. It also supports enumerated types, unnamed types, and tuples.

3.2 Problem specification

A simplified version of the well known Knapsack problem is chosen for its possible familiarity to the reader and its simple problem specification in Essence.

Given a collection of objects, each with an associated weight and value, find a subset of

these objects such that the total weight of the selected items is less than that we can carry and the total value is greater than a given amount.

An optimisation variant of this problem might be more interesting: instead of selecting subset of items such that the total value is greater than a given `minValue`, the problem might be to maximise the total value. Indeed, this variant of the problem is also very easy to specify in Essence, however it does not add a lot of value to our working example.

Another possible extension to this problem can be the addition of volume constraints. In addition to limiting the total weight of selected items, the total volume of the selected items can also be limited. This constraint can be expressed, and will be handled, very similarly to the weight constraint. Hence it is left out.

let us start by describing the problem specification Figure 3.1 for the Knapsack Problem. Line `1` is the language declaration; it tells Conjure that this file is written using the 1.3 version of the Essence language. This line will be common to all problem specifications given throughout the thesis.

Lines 3 to 6 are `given` statements. A `given` statement is used to declare a problem parameter. Problem specifications written in Essence can be parameterised by problem data and the data can also use the rich set of types and domains available in Essence. Line 3 declares `object` to be an enumerated domain, whose members will be given in a parameter file. Line 4 declares two parameters `weight` and `value`; both of which are total mappings from elements of the enumerated type `object` to positive integers. Line 5 declares two parameters `maxWeight` and `minValue`; both of which are positive integers. Line 6 declares the last parameter, `knapsackSize`, the exact number of objects our knapsack can carry.

Line 8 is the declaration of the only decision variable in the problem specification. Here, `knapsack` is a set variable. It needs to contain a fixed number of elements in it, and each element needs to be an `object`.

The constraints of the problem are given on lines 10 to 12. Here, `weight(i)` is a function application, it will look the weight of object `i` in the parameter function. Similarly for `value(i)`. The specification of the constraints also contain a quantification over the

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given weight, value : function (total) item --> int(1..),
6  given maxWeight, minValue : int(1..)
7  given knapsackSize : int(1..)
8
9  find knapsack : set (size knapsackSize) of item
10
11 such that
12     maxWeight >= sum i in knapsack . weight(i),
13     minValue  <= sum i in knapsack . value(i)
```

Figure 3.2: After enumerated domains are replaced with integer domains.

values of a decision variable set. This is an unusual feature of Essence, generally CP modelling languages do not allow modellers to write quantified expressions involving decision variables.

The following sections will show how Conjure produces an output model starting from this problem specification step by step.

3.3 Handling declarations with enumerated domains

A simple transformation is one where Conjure changes each enumerated domain into an integer domain. This is required because the output language, Essence′ does not support enumerated domains. In this example, the values of the enumerated domain will be given in a parameter file. We change the problem specification to require a single integer as a parameter, that is the number of values in the enumerated domain. This way, declarations with an enumerated domain can be replaced by integer domains. The result of this transformation is given in Figure 3.2.

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given weight, value : function (total) item --> int(1..),
6  given maxWeight, minValue : int(1..)
7  given knapsackSize : int(1..)
8
9  find knapsack : set (size knapsackSize) of item
10
11 given weight_1D : matrix indexed by [item] of int(1..),
12 given value_1D  : matrix indexed by [item] of int(1..),
13 find knapsack_Occr: matrix indexed by [item] of bool
14
15 such that
16     maxWeight >= sum i in knapsack#Occr . weight#1D(i),
17     minValue  <= sum i in knapsack#Occr . value#1D(i),
18     knapsackSize = sum i : item . toInt(knapsack_Occr[i])
```

Figure 3.3: After representation selection, using the `Occr` representation.

3.4 Choosing representations

The abstract types of Essence are not available in Conjure's output language Essence′; nor in any other existing constraint modelling language. A very important aspect of automated modelling with Conjure is *type refinement*. In our running example, there is only one decision variable, `knapsack`, a set variable. In addition, there are two parameters with abstract domains: `weight` and `value` both having function domains.

Conjure has at least one representation option for each abstract domain. One possible representation for a total function is using a one-dimensional matrix. A possible representation for a set variable is using an *occurrence* matrix, a matrix of Boolean variables for every value that can be in the set. A `true` assignment indicates membership in the set. Another possible representation for a set variable is using an *explicit* matrix, a matrix with a slot for each value in the set.

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given maxWeight, minValue : int(1..)
6  given knapsackSize : int(1..)
7
8  given weight_1D : matrix indexed by [item] of int(1..),
9  given value_1D  : matrix indexed by [item] of int(1..),
10 find knapsack_Occr: matrix indexed by [item] of bool
11
12 such that
13     maxWeight >= sum i : item , knapsack_Occr[i] . weight_1D[i],
14     minValue  <= sum i : item , knapsack_Occr[i] . value_1D[i],
15     knapsackSize = sum i : item . toInt(knapsack_Occr[i])
```

Figure 3.4: After expression refinement, using the `Occr` representation.

3.4.1 Using the Occurrence representation

Choosing the `1D` representation for both function domains, and the `Occr` representation for the set domain, we reach the intermediate problem specification given in Figure 3.3.

Here, lines 11 to 13 are newly added: they are the concrete representations of original declarations with abstract domains. Line 18 is also newly added, this line is posting a new constraint to make sure the concrete representation of a set using a Boolean matrix will contain exactly as many elements as the original set variable. Such constraints are called *structural constraints* and they are added to the model when needed by the representation.

Another important point to notice is *representation markers* attached to use sites of abstract decision variables and parameters. These markers are inserted at this phase to direct *expression refinement* at a later phase.

Three new declarations are added in Figure 3.3. These are *representations* of abstract decision variables in the original problem specification. At this point, the rest of the problem specification is still written in terms of the original abstract decision variables. The next step, expression refinement, will rewrite expressions using representations of decision variables instead of the original abstract decision variables. When every expression

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given weight, value : function (total) item --> int(1..),
6  given maxWeight, minValue : int(1..)
7  given knapsackSize : int(1..)
8
9  find knapsack : set (size knapsackSize) of item
10
11 given weight_1D : matrix indexed by [item] of int(1..),
12 given value_1D  : matrix indexed by [item] of int(1..),
13 find knapsack_Expl: matrix indexed by [int(1..knapsackSize)] of item
14
15 such that
16     maxWeight >= sum i in knapsack#Expl . weight#1D(i),
17     minValue  <= sum i in knapsack#Expl . value#1D(i),
18     allDiff(knapsack_Expl),
19     forAll i : int(1..knapsackSize-1) .
20         knapsack_Expl[i] < knapsack_Expl[i+1]
```

Figure 3.5: After representation selection, using the `Expl` representation.

involving a reference to an abstract decision variable is rewritten, abstract decision variables are removed from the problem specification.

The complete CP model after expression refinement is given in Figure 3.4. Lines 13 and 14 are changed in comparison to the previous step. Function application for `1D` representation of total functions is rewritten into a simple matrix dereference. The refinement of the quantified expression is slightly more involved though: it requires creating a new quantified expression over a finite integer domain. Moreover a *guard* is added to make sure the total sum only contains weights of those objects that are in the set.

All problem constraints are rewritten to use concrete versions of abstract decision variables and parameters. At this point the decision variable `knapsack`, the parameters `weight` and `value` are not referenced in any part of the problem specification. Hence, they can be deleted.

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given maxWeight, minValue : int(1..)
6  given knapsackSize : int(1..)
7
8  given weight_1D : matrix indexed by [item] of int(1..),
9  given value_1D  : matrix indexed by [item] of int(1..),
10 find knapsack_Expl: matrix indexed by [int(1..knapsackSize)] of item
11
12 such that
13     maxWeight >= sum i : int(1..knapsackSize) .
14                          weight_1D[knapsack_Expl[i]],
15     minValue  >= sum i : int(1..knapsackSize) .
16                          weight_1D[knapsack_Expl[i]],
17     forAll i : int(1..knapsackSize-1) .
18         knapsack_Expl[i] < knapsack_Expl[i+1]
```

Figure 3.6: After expression refinement, using the `Expl` representation.

3.4.2 Using the Explicit representation

Again, choosing the `1D` representation for both function domains, and the `Expl` representation for the set domain, we reach the intermediate problem specification given in Figure 3.5.

Similar to the previous section, the concrete representations of original declarations with abstract domains and structural constraints are added. Also, each use site of abstract declarations variables and parameters are marked with the name of the representation to direct expression refinement.

An important observation here is about symmetry breaking. The structural constraint on line 18 is enough to maintain the distinctness requirement of the original abstract domain. However, using an `allDiff` we introduce symmetry, that is there are multiple assignments to the matrix domain which all represent the same set value. In order to break the symmetry[1], we introduce another structural constraint given on lines 19 and 20. Adding this constraint

[1] In general, a set being represented using the Explicit representation does not have to contain integers in it. Symmetry breaking as applied by Conjure in its full generality is explained in Chapter 8.

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given weight, value : function (total) item --> int(1..),
6  given maxWeight, minValue : int(1..)
7  given knapsackSize : int(1..)
8
9  find knapsack : set (size knapsackSize) of item
10
11 given weight_1D : matrix indexed by [item] of int(1..),
12 given value_1D  : matrix indexed by [item] of int(1..),
13 find knapsack_Expl: matrix indexed by [int(1..knapsackSize)] of item
14 find knapsack_Occr: matrix indexed by [item] of bool
15
16 such that
17     maxWeight >= sum i in knapsack#Expl . weight#1D(i),
18     minValue  <= sum i in knapsack#Occr . value#1D(i),
19     knapsackSize = sum i : item . toInt(knapsack_Occr)
20     forAll i : int(1..knapsackSize-1) .
21         knapsack_Expl[i] < knapsack_Expl[i+1],
22     knapsack#Expl = knapsack#Occr
```

Figure 3.7: After representation selection, in a channelled model.

renders the `allDiff` redundant, so we do not actually add it to the model.

The complete CP model after expression refinement is given in Figure 3.6. Lines 13 to 16 are changed in comparison to the previous step. Similar to what happened in the `Occr` representation, function application for `1D` representation of total functions is rewritten into a simple matrix dereference. The refinement of the quantified expression is turned into a new quantified expression over a finite integer domain. This time the quantified expression does not require a *guard*, and it uses a simple matrix dereference to access an object in the set.

3.4.3 Channelled models

If an abstract decision variable or parameter is used in multiple contexts in a problem specification, a different representation may be chosen for each use site. In cases when

```
1  language Essence 1.3
2
3  given item_Count : int(1..)
4  letting item be domain int(1..item_Count)
5  given maxWeight, minValue : int(1..)
6  given knapsackSize : int(1..)
7
8  given weight_1D : matrix indexed by [item] of int(1..),
9  given value_1D  : matrix indexed by [item] of int(1..),
10 find knapsack_Expl: matrix indexed by [int(1..knapsackSize)] of item
11 find knapsack_Occr: matrix indexed by [item] of bool
12
13 such that
14     maxWeight >= sum i : int(1..knapsackSize) .
15                          weight_1D[knapsack_Expl[i]],
16     minValue  <= sum i : item , knapsack_Occr[i] . value_1D[i],
17     knapsackSize = sum i : item . toInt(knapsack_Occr)
18     forAll i : int(1..knapsackSize-1) .
19         knapsack_Expl[i] < knapsack_Expl[i+1],
20     forAll i : int(1..knapsackSize) .
21         knapsack_Occr[knapsack_Expl[i]],
22     forAll i : item , knapsack_Occr[i] .
23         exists j : int(1..knapsackSize) .
24             knapsack_Expl[j] = i
```

Figure 3.8: After expression refinement, in a channelled model.

multiple representations are selected for the same decision variable, *channelling constraints* are required to make sure different versions of the same abstract decision variable get assigned to the same value in every solution.

In the running example, the decision variable `knapsack` occurs in two different contexts. Conjure can select different representations for it in different contexts. Figure 3.7 shows the intermediate problem specification after representation selection for a channelled model.

In this alternative model, both concrete representations are added to the model at the same time. Notice the representation markers, the first constraint on line 17 is marked to use the Explicit representation for `knapsack`, whereas the second constraint on line 18 is marked to use the Occurrence representation.

Structural constraints for both representations are added to the model (lines 19 to 21).

Moreover, on line 22, a channelling constraint is added. The channelling constraint is simply an equality constraint on two set variables, which happen to refer to the same set variable but represented differently. Luckily, this is the only specific treatment Conjure needs to do for channelling constraints, from here on they will be refined in the same way as the other constraints in the original problem.

Highlighted in Figure 3.8 is the refinement of the channelling constraint.

3.5 Summary

This chapter demonstrated Conjure's operation on a concrete problem specification without going into details of *how* each step actually works. Later chapters of the thesis will describe how Conjure is designed and implemented in its full generality.

Chapter 4

Design and Architecture

This chapter is describes the design and architecture of Conjure. It is divided into two parts. In the first part we look from the outside-in and see the components of a constraint programming tool-chain, motivations behind having such a tool-chain, and the part Conjure plays in enabling an effective constraint programming system. In the second part we focus on Conjure's internals and see how it operates, the pipeline of operations, and how it uses a rule language to describe most of its transformations.

4.1 Outside-In: The Tool-Chain

This section describes Conjure in the context of a tool-chain comprised of Conjure itself, SavileRow the instance level constraint modelling assistant, and the constraint solver Minion.

4.1.1 Minion and its input language

Minion[Gen+06] is a powerful constraint solver. It takes as input a problem instance model written in its specific input language. The input language cannot encode problem-class models, it is very low level and it closely matches solvers internals. A model consists of declaration of decision variables, posting constraints and optionally the objective value. Domains in Minion are basically integers with finite domains. The solver provides four

different implementations of domains for decision variables: `BOOL`, `DISCRETE`, `BOUND`, and `SPARSEBOUND`. Minion's type system does not discriminate between different kinds of domains; however, some constraints only work on decision variables with certain kinds of domains.

`BOOL` Boolean domains. Decision variables with Boolean domains are used very commonly for logical expressions.

`DISCRETE` Finite integer domains specified by a lower and an upper bound. Propagation and search can prune individual values from the domain of a decision variable with a `DISCRETE` domain. Namely, arbitrary holes can be created in the domains of the variables.

`BOUND` Finite integer domains specified by a lower and an upper bound. Only the bounds of the domain are maintained. Propagation and search can only prune from the lower or the upper bound from the domain of a decision variable with a `BOUND` domain. Namely, holes cannot be put in the domains of the variables.

`SPARSEBOUND` Finite integer domains specified by listing all the elements of the domain, but only the upper and lower bounds of the domain may be updated during search. Propagation and search can only prune from the lower or the upper bound from the domain of a decision variable with a `SPARSEBOUND` domain. Namely, any holes in the domain must be there at the time of declaration and holes cannot be created during the solving process.

The constraint language of Minion also closely follows the internals of the solver. Constraints are mostly *flat*. In this context, a constraint expression is flat if it does not take other constraints as arguments. However, Minion still has some non-flat constraints. An indispensable non-flat constraint is constraint reification via the `reify` constraint. Using `reify`, we can relate the truth value of a constraint to an auxiliary boolean decision variable

and use this decision variable in the rest of the model instead of the constraint itself. Constraint reification is not the only non-flat constraint, MINION also has `watched-or` and `watched-and` which take a list of constraints as arguments and post logical-or and logical-and conditions on the arguments respectively.

Some constraints have multiple alternative implementations; such as `alldiff` and `gacalldiff`. Generally, different versions of the same constraint enforce different levels of consistency.

Writing MINION input by hand is not practical for a few reasons. First is the lack of parameterised models; generally a user wants to model a class of problems instead of a single instance. Second is the lack of abstraction; a user will have to be familiar with a lot of low level details about MINION before they can write correct models. In order to write *good* models, they will have to make a lot of low level decisions possibly without understanding all the trade-offs. Third is the verbosity of the input language. More often than not, CP modelling requires experimentation. Working with such a verbose language limits the ability to experiment. For instance, changing the viewpoint of the model or changing how a constraint is formulated requires rewriting almost the whole model.

4.1.2 SAVILEROW and ESSENCE′

SAVILEROW is a constraint modelling assistant. It takes as input a CP model written in ESSENCE′. ESSENCE′ is a CP modelling language that allows encoding of problem-class models, and allows constraint expressions and the objective to contain non-flat expressions. Using non-flat expressions let the modeller write mathematical equations using arithmetic and logical operators and use the results of intermediate expressions as parts of larger expressions. In this setting, constraint reification becomes implicit. In effect, the truth value of a constraint expression can be treated as a Boolean value in every context where a simple Boolean literal can be used. As an example, one feature of ESSENCE′ which assists CP modellers is the overloading of the matrix indexing operator to use the `element` constraint if needed, and use a simple matrix dereference when the argument is not a decision variable.

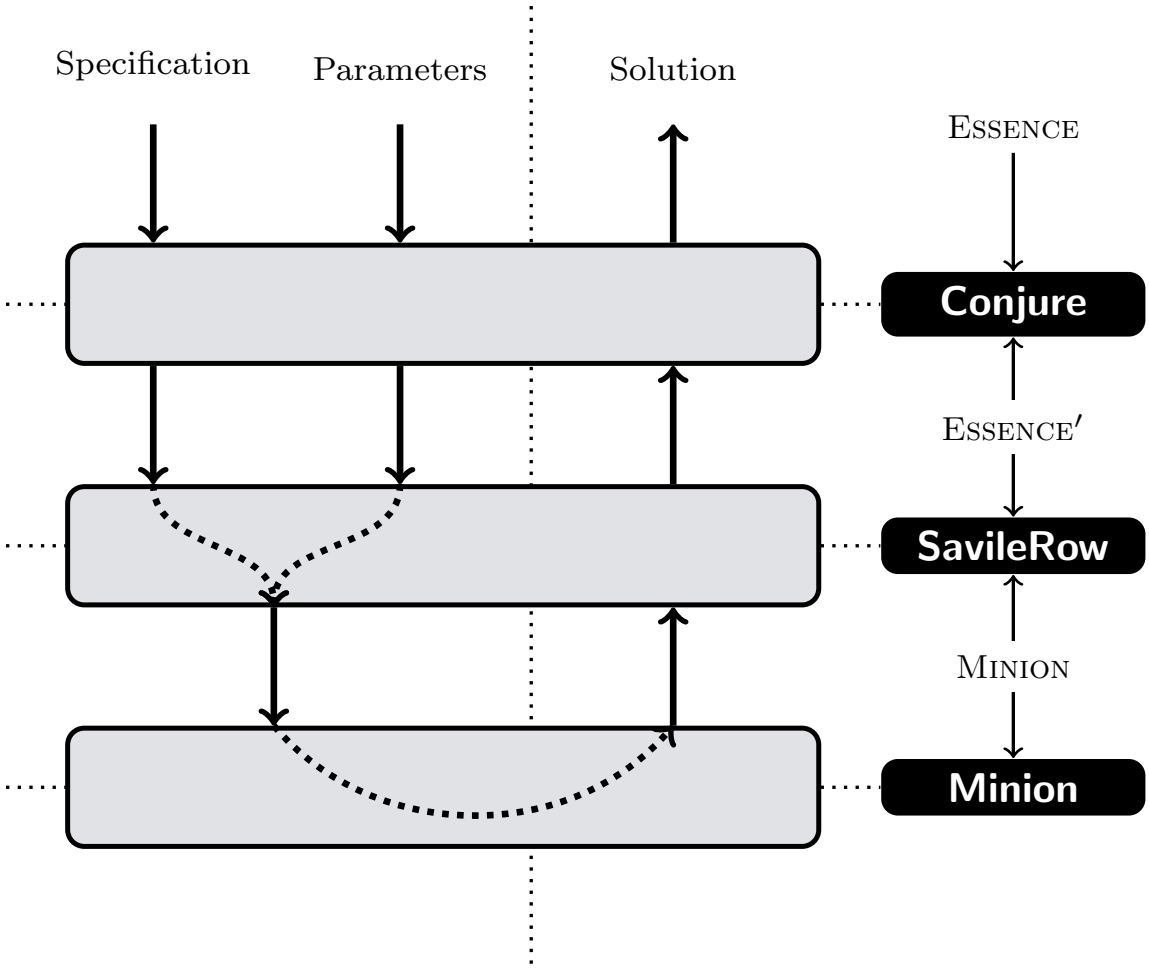


Figure 4.1: Automated Constraint Modelling Tool-Chain

Although Essence′ adds more expressivity to the expression language, it does not add richer domains. In Essence′ the domains of decision variables are essentially finite integer ranges, similar to domains in Minion's input language.

Essence′ provides a level of abstraction and expressivity that is comparable to that of other CP modelling languages. CP modellers can use Essence′ to write problem class models, then use SavileRow to instantiate parameters and generate a Minion instance model, solve it using Minion and finally use SavileRow to translate solutions of Minion to solutions represented using Essence′. Thanks to the higher level of abstraction provided by

Essence′ (in comparison to Minion), most of the shortcomings of using Minion's input directly are solved. Some low level decisions like which kind of domain to use for a decision variable are made automatically by SavileRow. Refactoring models is relatively easier because the language is less verbose.

SavileRow does not merely translate Essence′ to Minion.It also employs powerful micro-modelling optimisations. These optimisations include common subexpression elimination (CSE), constant folding, rearranging weighted sums and etc. Most of these optimisations become more effective after parameter instantiation, because more information is available at this point. SavileRow can also target multiple solvers. Essence′ is a solver independent CP modelling language, and SavileRow is the translator which makes multiple solver accessible. We focus on the Minion backend in this thesis because it is the most mature backend, and working with multiple solvers does not add value to the key contributions of this thesis.

4.1.3 Conjure and Essence

Conjure is an automated CP modelling tool. Instead of requiring the user to produce a concrete CP model, it takes as input an abstract problem specification written in Essence; and generates automatically Essence′ models as output. Essence is a high-level problem specification language. It provides a rich set of built-in domains and domain constructors; such as sets, multi-sets and functions. Decision variables can have these domains so as to precisely encode what they *mean*. Without the need to *model* these complex domains via multiple decision variables with simpler domains and relating them in the rest of the problem specification via additional constraints. The full set of domains and domain constructors in Essence are given in Table 4.1.

Another feature of Essence is to help enable succinct problem specifications is domain attributes. Attributes further restrict (i.e. make precise) an abstract domain, so the user of Essence does not need to use constraints to achieve the desired effect. For instance, a set variable can have a `minSize` attribute attached to it, which which make sure the values of a

decision variable with this attribute can only be sets containing at least a given number of elements.

Essence is statically typed. For a decision variable or a parameter, stripping attributes from the domain leaves us with the type. Essence also has a rich collection of operators to write expressions of using smaller expressions with abstract types. For example, for functions, there is an `inverse` operator, which makes sure two functions are inverses of each other. For relations, relation projection lets a use create a relation of smaller arity while fixing some of the components to a specific value. The complete set of operators in Essence is given in Table 4.3.

In addition to those listed in the figure, Essence provides a rich collection of quantified expressions. Many CP modelling languages allow quantifiers such as `forAll`, `exists`, and `sum` to be used over finite integer ranges. This is a very powerful feature and Essence (as well as Essence′) have these constructs.

Generally, quantified variables take values from the given integer range. The value of the whole quantified expression is determined by unrolling the quantified expression and aggregating each item of this unrolling depending on the quantifier keyword. Conjunction, disjunction and addition are used respectively for `forAll`, `exists` and `sum` quantifier keywords.

In addition to this conventional understanding of quantified expressions, Essence allows any domain to be used instead of a simple integer range in quantified expressions. This means, a constraint expression can be written `forAll` values of a finite multi-set domain. Essence also allows quantification over arbitrary set and multi-set decision variable expressions.

For example, the following constraint posts the condition that every element in a set decision variable `x` has to be even if that element is strictly greater than `10`.

```
forAll i in x , i > 10 . i % 2 = 0
```

Domain	Kind	Handling
`bool`	Concrete	Kept unchanged
`int`	Concrete	Kept unchanged
enumerated	Abstract	Mapped to integers
unnamed	Abstract	Mapped to integers
`tuple (` $\tau_1, ..., \tau_i, ...$ `)`	Abstract	Separated into components
`set of` τ	Abstract	Represented
`mset of` τ	Abstract	Represented
`function` τ_1 `->` τ_1	Abstract	Represented
`relation of (` $\tau_1, ..., \tau_i, ...$ `)`	Abstract	Represented
`partition from` τ	Abstract	Represented

Table 4.1: Domains in Essence

Domain	Attributes
`set of` τ	`size`, `minSize`, `maxSize`
`mset of` τ	`size`, `minSize`, `maxSize`, `minOccur`, `maxOccur`
`function` τ_1 `->` τ_1	`total`, `partial`, `injective`, `surjective`, `bijective`
`relation of (` $\tau_1, ..., \tau_i, ...$ `)`	`size`, `minSize`, `maxSize`
`partition from` τ	`size`, `minSize`, `maxSize`, `regular`, `complete`, `partSize`, `minPartSize`, `maxPartSize`, `numParts`, `minNumParts`, `maxNumParts`

Table 4.2: Domains in Essence

4.1.3.1 Explicit guards

A quantified expression has 3 main parts. The first part is *preamble* where a quantifier keyword -`forAll`, `exists`, `sum`- is given together with a quantified variable declaration. The second part is optional and contains a `guard`. Guard is a boolean expression, and it is written using a comma after the preamble of a quantified expression. Conceptually, guards can be viewed as indicating whether the body of a quantified expression is active or not for each value of the quantified expression. The third part is the *body* part. In the above example `forAll i in x` is the preamble, `i > 10` is the guard, and `i % 2 = 0` is the body of the quantified expression.

Domain	Attributes
`set of` τ	`union`, `intersect`, \ *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\|x\|$ *(cardinality)*, `in`, `max`, `min`
`mset of` τ	`union`, `intersect`, \ *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\|x\|$ *(cardinality)*, `in`, `freq`, `hist`, `max`, `min`
`function` τ_1 `->` τ_1	`union`, `intersect`, \ *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\|x\|$ *(cardinality)*, `defined`, `range`, *function application*, `inverse`, `preImage`
`relation of (` $\tau_1, ..., \tau_i, ...$ `)`	`union`, `intersect`, \ *(difference)*, `subset`, `subseteq`, `supset`, `supseteq`, $\|x\|$ *(cardinality)*, *relation application*, *relation projection*
`partition from` τ	$\|x\|$ *(cardinality)*, `together`, `apart`, `participants`, `parts`, `party`

Table 4.3: Operators in Essence

4.1.4 The tool-chain

The complete tool-chain is given in Figure 4.1. An Essence problem specification is input to the Conjure system, which employs a system of modelling transformations to refine the specification into a concrete CP model in the Essence′ language. A CP model typically describes a parameterised problem class. An instance of the class is obtained for input to the constraint solver by giving values for the parameters. Essence provides the same facility, allowing the specification of problem classes. The refinement of problem specification and parameter values is separated in the tool-chain, as shown in the figure. This allows the user to solve multiple instances from the same problem class while only performing model refinement once.

The SavileRow system accepts an Essence′ model and corresponding parameter values. It instantiates the model and transforms it into the input suitable for the Minion constraint solver. SavileRow is able to produce output suitable for other constraint solvers, but this thesis will use the Minion backend only. After Minion has solved the problem instance, SavileRow translates the solution back into Essence′. Conjure then translates the Essence′ solution into a solution to the original Essence problem specification for presentation to the

user.

Conjure can generate multiple models for a given problem. Selecting a good model from the available candidates remains a challenging task.

4.2 Inside-Out: Conjure's Inner Workings

This section describes Conjure's inner workings by explaining each step of its pipeline in a subsection. Examples are used when necessary to describe how a certain step operates. Essence hides CP modelling complexity very well, a concise problem specification often produces quite large Essence′ models. It is because of this reason why small Essence specifications or fragments of larger specifications are used as examples.

4.2.1 Abstract Syntax Tree

Conjure does not store and manipulate Essence problem specifications in source form. Inputs are *parsed* into an Abstract Syntax Tree (AST) representation and outputs are *pretty-printed* to produce the human readable textual representation.

At the top level, an Essence specification is a list of statements. Some statements introduce new declarations of decision variables or parameters, for this reason the order of statements is important. Internally, Conjure stores a singly-linked list of statements and each statement is stored in a data structure that is the AST of Essence. Conjure's output – CP models in Essence′ – are also stored using the same representation. The only difference between the two languages in this regard is the types of AST nodes they support.

Due to the complexity of both Essence and Essence′ and to keep the internal representation as generic as possible Conjure uses a simple rose-tree for the AST. Each node is *tagged* by a marker to differentiate kind of AST node. For example, the simple problem specification given in Figure 4.2 declares 3 decision variables each having the identical domain `int(1..3)` and two constraints. The comma-separated list of identifiers on line 3 is just syntax, writing that line is equivalent to writing 3 separate `find` statements and

```
1 language Essence 1.3
2
3 find x,y,z: int(1..3)
4
5 such that
6     y + 2 = z,
7     allDiff([x,y,z])
```

Figure 4.2: A simple Essence specification

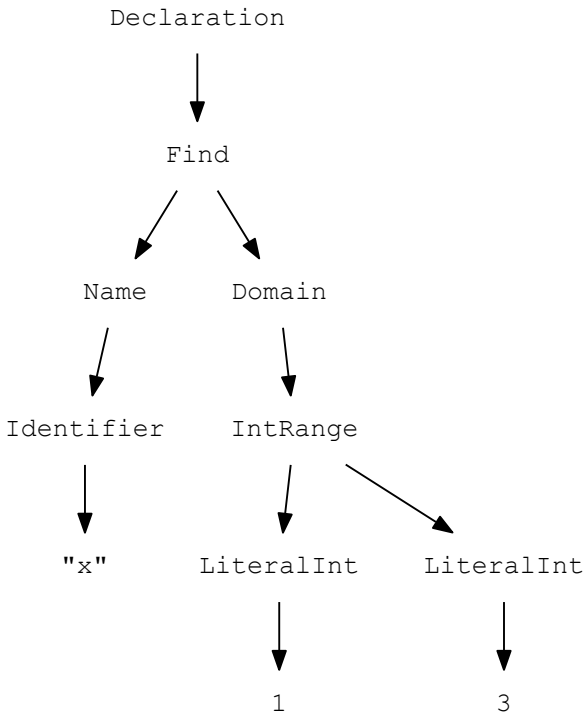


Figure 4.3: AST for a declaration

replicating the domain 3 times. The AST representation for one of these declarations is given in Figure 4.3. AST representations of the constraints are given in Figures 4.4 and 4.5.

The AST permits easy traversals to do various lookups and modifications to the stored program. All operations described in the rest of this chapter work on the AST representation.

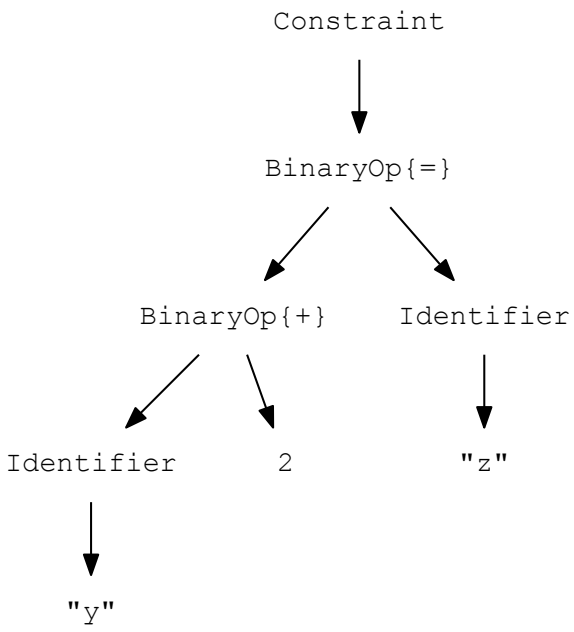


Figure 4.4: AST for an all-different constraint

Moreover, Chapter 5 describes how a rule language is implemented using a generic matching mechanism together with the parsing and pretty-printing functions to translate between Essence syntax and the AST representation.

4.2.2 Type-checking

After parsing Essence input and creating an equivalent AST representation for it, Conjure proceeds to type-checking. Essence is statically typed and type-incorrect input need to be rejected before further processing.

Essence specification are a list of statements. The type-checking an Essence specification is comprised of type-checking each statement in order. During this process, an symbol table containing type information for each identifier is kept as state. The state is implemented using a stack of identifier-type pairs; using a stack, entering a new binding scope can be

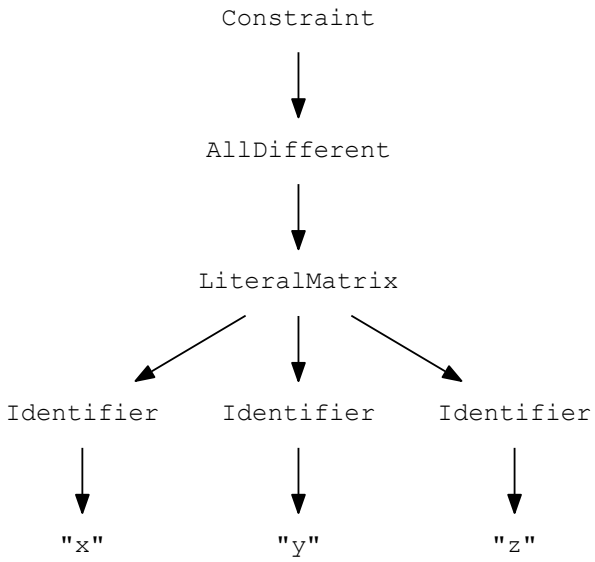


Figure 4.5: AST for another constraint

implemented using a stack-push operation, and coming out of a binding scope can be implemented using a stack-pop.

Algorithm 4.1: Type-checking an Essence problem specification

```
Algorithm tcEssence(statements)
    symbolTable ← initialiseStack
    foreach s in statements do
        symbolTable ← tcStatement(s,symbolTable)
    end
    return
```

This internal representation is very generic, i.e. it lets us represent invalid ASTs as well as valid ones. In particular, it can represent type-incorrect Essence expressions. Essence is statically typed, and type-incorrect Essence input need to be rejected by Conjure. In order to achieve this, Conjure employs a separate phase to type-check the Essence represented using the AST (Algorithm 4.1). In Essence, the type of each domain is easily determined

Algorithm 4.2: Type-checking an individual statement

```
Algorithm tcStatement(statement, symbolTable)
    switch statement do
        case Find(name,domain)
            tcDeclaration(name, domain, symbolTable)
        end
        case Given(name,domain)
            tcDeclaration(name, domain, symbolTable)
        end
        case LettingExpr(name,expression)
            ty ← tcExpression(expression)
            symbolTable ← push(⟨name,ty⟩,symbolTable)
            return symbolTable
        end
        case LettingDomain(name,domain)
            ty ← tcDomain(domain)
            symbolTable ← push(⟨name,ty⟩,symbolTable)
            return symbolTable
        end
        case Where(expression)
            ty ← tcExpression(expression, symbolTable)
            if ty != bool then
                reportError(statement)
            end
            return symbolTable
        end
        case SuchThat(expression)
            ty ← tcExpression(expression, symbolTable)
            if ty != bool then
                reportError(statement)
            end
            return symbolTable
        end
        case Objective(expression)
            ty ← tcExpression(expression, symbolTable)
            if ty != int then
                reportError(statement)
            end
            return symbolTable
        end
    endsw
```

Algorithm 4.3: Type-checking declaration statements

Algorithm `tcDeclaration`(*name, domain, symbolTable*)
  ty ← tcDomain(domain)
  symbolTable ← push(⟨name,ty⟩,symbolTable)
  return symbolTable

by removing attributes from a domain. Each decision variable or parameter declaration will have a domain, and the type of the newly declared identifier will be that of its domain (Algorithm 4.3). Expressions are built using predefined operators and typing rules for each predefined operator is know; hence, the type of any expression can be calculated using these typing rules (Algorithm 4.2). Each top level constraint has to be of type Boolean and the objective expression has to be of type integer to be type-correct.

4.2.3 Input Validation

Input validation phase comes after type-checking. It validates the input by checking attributes of abstract decision variables for inconsistencies and adding *implicit instantiation conditions* if needed.

In Essence explicit instantiation conditions are given by `where` statements. A `where` statement contains a boolean expression which is checked at instantiation time, and hence cannot contain expressions that depend on the values of decision variables. In addition to the explicit ones, a specification also imposes implicit instantiation conditions as follows:

- A declaration of the form "`letting` *id* `be new type of size` *e*" imposes the instantiation condition that $e \geq 1$
- Integer ranges used in matrix indices need to be non-empty. For example using the range `int(a..b)` as a matrix index imposes the instantiation condition that $a \leq b$.
- Use of annotations of certain forms imposes the instantiation condition $e \geq 0$:

  (`size` *e*), (`maxSize` *e*), (`minSize` *e*), (`partSize` *e*), (`minPartSize` *e*), (`maxPartSize` *e*), (`numParts` *e*), (`minNumParts` *e*), (`maxNumParts` *e*), (`minOccur` *e*), (`maxOccur` *e*).

- Constructing a set by giving each of its elements, imposes the instantiation condition that each value is distinct.
- Constructing a function by giving each of its constituent mappings, imposes the instantiation condition that each value is mapped to at most one other value.
- Similarly, constructing a partition by giving each of its parts, imposes the instantiation condition that the parts are disjoint.

4.2.4 Representation selection for a single declaration

In Essence, a decision variable or a problem parameter is defined by giving it a name – a unique identifier that is previously not used in the problem specification – and a domain. The domain of either kind of declaration can either be concrete or abstract. Concrete domains are domains which are also supported by Essence′: `bool`, `int`, and `matrix` domains containing elements of `bools` or `ints`. In Essence′, matrix domains cannot contain other matrix domains, but they can be multi-dimensional. Declarations with concrete domains can be kept unchanged in Conjure's output, thus they do not need to be *represented* in some other way.

On the other hand, declarations with abstract domains cannot occur in Conjure's output, thus they need to be represented using simpler declarations. Representation selection in Conjure takes in as argument an abstract domain and produces a list of candidate representation options. Each option consists of three components: a name for the representation, a new domain to be used instead of the original abstract domain and optionally structural constraints to be posted.

Structural constraints are needed to make sure the replacement domain does not contain values that the original abstract domain does not. For example, a function decision variable between two integer domains can be represented using a 2-dimensional matrix of Boolean variables. If no constraints are posted on this representation, values which are not valid mathematical functions will be values of this domain. As an example, let us look at

Figure 4.6 and Figure 4.7. Here, the decision variable `f` is represented using a 2-dimensional matrix of Booleans, and a structural constraint is added. It is important to notice that the replacement domain will contain values which are not valid mathematical functions without this constraint.

Structural constraints are also helpful when representing domains with additional attributes on them. In the previous example, `f` did not have any attributes, so the function variable could be a partial function. However, Essence lets us decorate domains with additional information via attributes. In such cases, structural constraints are used to *refine* the replacement domain to only contain values in accordance with the attributes. Figure 4.8 lists a function domain with a `total` attribute, and Figure 4.9 is the refinement of it. Notice that the refinement of the same function domain without the attribute used the exacts same replacement domain, however, the structural constraint is different. The structural constraint without `total` only made sure that there is at most 1 mapping for each value in the range over which the function is defined; whereas with the `total` keyword, there has to be exactly 1 mapping for each value.

Finally, structural constraints are used to break symmetry that is introduced by representing a domain. As an example, let us look at Figure 4.10 and Figure 4.11. Here, the abstract decision variable `s` is represented using a 1-dimensional matrix of 4 integers. In order to preserve the meaning of the original problem specification, an all-different constraint has to be posted as a structural constraint, since mathematical sets cannot contain duplicate items. However, if we use an all-different constraint, we will be introducing *modelling symmetry*. Instead of the all-different, Conjure generates structural constraints to make sure values of `s_Explicit` are all-increasing. This breaks the symmetry that is introduced due to modelling. See 8 for more details on how symmetry breaking works in Conjure.

Posting a structural constraint rules out such values from the domain.

```
language Essence 1.3

find f : function int(1..3) --> int(1..3)
```

Figure 4.6: A simple Essence problem specification containing only one abstract decision variable, `f`, with a function domain.

```
language Essence 1.3

find f_Matrix2D : matrix indexed by [int(1..3), int(1..3)] of bool

such that
    forAll i : int(1..3) .
        (sum j : int(1..3) . toInt(f_Matrix2D[i,j])) <= 1
```

Figure 4.7: A representation option for the decision variable listed in Figure 4.6

```
language Essence 1.3

find f : function (total) int(1..3) --> int(1..3)
```

Figure 4.8: A variation of Figure 4.6 with an additional attribute on the domain.

```
language Essence 1.3

find f_Matrix2D : matrix indexed by [int(1..3), int(1..3)] of bool

such that
    forAll i : int(1..3) .
        (sum j : int(1..3) . toInt(f_Matrix2D[i,j])) = 1
```

Figure 4.9: A representation option for the decision variable listed in Figure 4.8.

```
language Essence 1.3

find s : set (size 4) of int(0..9)
```

Figure 4.10: A simple Essence problem specification containing only one abstract decision variable, `s`, with a set domain.

```
1 language Essence 1.3
2
3 find s_Explicit : matrix indexed by [int(1..4)] of int(1..9)
4
5 such that
6     forAll i : int(1..3) . s_Explicit[i] < s_Explicit[i+1]
```

Figure 4.11: A representation option for Figure 4.10

4.2.5 Identifier Regions

Each identifier referring to a decision variable or a parameter declaration can be annotated with region information.

An identifier, and the declaration referenced by the identifier, will be represented in the same way by Conjure.

However, the same declaration can be represented differently if they are in different regions.

Motivation: To enable the use of different representations of a single decision variable in the same CP model. If a model uses multiple representations of a single decision variable, channelling constraints are generated automatically by Conjure.

Also to have a mechanism enabling finer grained control of how many different representations of a single decision variable can be used in a single model. Assigning *related* occurrences of the same decision variable the same region will force Conjure to use the same representation for all of those.

The most variation will come when each occurrence is assigned a unique region. The goal of Conjure at this stage is to systematically explore a diverse space of models, so it indeed assigns a separate region to each identifier.

let us extend the specification from Figure 4.6 with two constraints. In Figure 4.12, the first constraint uses function application to post the condition that the value `1` is mapped to the value `3`. The second constraint posts the condition that the cardinality of the set of values mapped to the value `2` should be at least `1`. Namely, in an assignment to `f`, there needs to

```
1 language Essence 1.3
2
3 find f : function (total) int(1..3) --> int(1..3)
4
5 such that
6     f(1) = 3,
7     |inverse(f,2)| >= 1
```

Figure 4.12: An Essence problem specification, which uses `f` twice.

be at least one value that is mapped to 2. The decision variable `f` can be represented in multiple ways by Conjure. The two obvious choices are using a one-dimensional matrix of three values or using a two-dimensional matrix of boolean values. Conjure automatically puts the two `f`'s into separate regions, so it can choose different representations for each occurrence of the decision variable `f` in the specification.

4.2.6 Representation selection for a complete Essence specification

Conjure selects a representation for each identifier that is referring to an abstract declaration. It can select different representations for different occurrences of the same identifier. If multiple representations are selected for the same declaration, Conjure generates an equality constraint between the two different representations; such constraints are called *channelling constraints*. The equality constraint is refined in the same way to other constraints in the problem specification, it is not handled specially.

A representation for an abstract declaration might use other abstract domains. An obvious example of this phenomena is nested domains, a `set of` τ where τ is an abstract domain will generate a matrix of τ domains. This domain will need further representation. Another example is when an abstract domain is represented using another abstract domain. For example `function` domains can be represented using `relation` domains with additional constraints on them.

Figure 4.13, gives an example problem specification where the two occurrences of the decision variable are annotated with different representation decisions. In addition to

```
language Essence 1.3

find f : function (total) int(1..3) --> int(1..3)
find f_Matrix1D : matrix indexed by [int(1..3)] of int(1..3)
find f_Matrix2D : matrix indexed by [int(1..3),int(1..3)] of bool

such that
    f#Matrix1D(1) = 3,
    |inverse(f#Matrix2D,2)| >= 1,
    // structural constraints for f_Matrix1D
    // structural constraints for f_Matrix2D
    f#Matrix1D = f#Matrix2D     // channelling constraint
```

Figure 4.13: Figure 4.12 annotated with representation decisions.

annotating each occurrence of `f`, Conjure also generates the new declarations for the selected representations. Structural constraints are generated if any are needed for the new representation and an equality constraint is added between the two representations of `f`.

4.2.7 Expression refinement

After a representation is chosen for each abstract declaration, expressions containing references to these declarations need to be *refined*. Such expressions are called abstract expressions.

Expression refinement is the process of replacing an expression with an equivalent expression. It is commonly used in Conjure to replace abstract expressions with concrete expressions. It typically takes multiple replacements to reach a final and fully concrete version of the original expression.

Abstract expressions can be contained in constraint statements, the objective statement if there is one, expressions on the right-hand side of letting statements, and expressions in where statements.

Conjure contains a collection of transformations which work on fragments of Essence expressions. These transformations do not necessarily operate on complete constraint expressions, they can operate on any sub-expression contained within larger expressions.

```
1 language Essence 1.3
2
3 find f : function (total) int(1..3) --> int(1..3)
4 find f_Matrix1D : matrix indexed by [int(1..3)] of int(1..3)
5
6 such that
7     f#Matrix1D(1) = 3
```

Figure 4.14: Showing the operation of expression refinement on parts of Figure 4.13.

For example, in an expression like `a union b = c` – where `a`, `b` and `c` are sets – an expression refinement transformation can rewrite the left-hand side without touching the rest of the expression at all.

Transformations either match a given expression or they do not. If a transformation matches an expression, it returns a rewrite of the original expression. Moreover, multiple transformations can match a single expression. When multiple transformations are applicable, this means there are multiple ways to *refine* the expression at hand. Conjure forks at this point and generates an output model using each option in turn.

Figure 4.14 shows the operation of expression refinement on parts of an Essence specification. On line 7, `f#Matrix1D(1)` is an abstract expression. It contains a reference to `f`, an abstract decision variable declaration. It uses the function-application syntax to refer to the value that is mapped to by value `1`.

Conjure's expression refinement phase will find only one applicable transformation in this case, and that transformation will replace the function-application operator to a simple matrix dereference operator. It will also change the parameter of the new operator, instead of `f` annotated with representation information, it will use the concrete refinement of `f`, `f_Matrix1D`. The result of applying this transformation will produce `f_Matrix1D[1] = 3` as the final constraint. At this point the abstract decision variable `f` is not being referred to at any point in the problem specification. When an abstract declaration is not used in the problem specification, it is said to be fully refined and can safely be removed from the specification (See Figure 4.15).

```
1 language Essence 1.3
2
3 find f_Matrix1D : matrix indexed by [int(1..3)] of int(1..3)
4
5 such that
6     f_Matrix1D[1] = 3
```

Figure 4.15: After refining the expression and removing the abstract declaration of Figure 4.14.

4.2.8 Partial evaluator

Conventional compilers for programming languages use partial evaluation for program optimisation. They use statically known data to reduce parts of the program at compile time. This process potentially makes the program run faster while producing identical results.

Partial evaluation has a much bigger potential in the context of CP modelling languages. Eliminating unnecessary constraints and decision variables can drastically improve the solving time of a CP model. The potential gain from partial evaluation gets larger directly proportional to the level of abstraction of the CP modelling language. It is an indispensable tool when operating on a problem specification language with abstract domains and powerful operators: eliminating one decision variable with a function domain eliminates multiple decision variables and constraints from the generated model.

For example, a problem specification like the one in Figure 4.16 contains a top level constraint which is trivially true. Conjure will transform this constraint into `x = x` first by eliminating the `union` with an empty set, and then remove the constraint from the model altogether.

The partial evaluator is also used by the *rule language* of Conjure, which is described in Chapter 5.

4.2.9 Enumerated types and Unnamed types

Essence offers enumerated types and unnamed types to use during problem specification.

Enumerated types are finite types and their members unique identifiers explicitly listed

```
1 language Essence 1.3
2
3 find x: set of τ
4 such that
5     x union {} = x,
6     ...
```

Figure 4.16: Partial evaluation can remove the constraint at line 5.

by the user when defining the type. When enumerated types are not provided by a modelling system, integers are generally used to represent enumerations. This encoding requires the user to maintain the mapping between items and integers externally to the system. In addition to helping reduce this burden from the user, supporting enumerated types internally comes with an added benefit: Enumerated types do not support all operators of integers. Most importantly, they do not support arithmetic operators, and defining a decision variable to have an enumerated type as a domain limits the user from making mistakes such as using the decision variable in a type-incorrect way. Enumerated types support equality and inequality checks. They also support checking for ordering using the common operators: <, <=, >, and >=. Element order follows declaration order.

Unnamed types, similar to enumerated types, are generally modelled using integers in most CP modelling languages. Essence offers unnamed types to use when members of a type are not named, i.e. they are freely interchangeable. They let the user avoid introducing unnecessary symmetry to the problem specification. For example, golfers in the Social Golfers Problem are interchangeable, the problem does not post any special constraints for specific golfers. Using unnamed types, Conjure has access to the information that golfers are interchangeable. Unnamed types only support equality and inequality checks.

Refinement of both enumerated types and unnamed types are simply mapping them to integers. Conjure uses a finite integer domain containing the correct number of elements to model enumerated types and unnamed types.

4.3 Summary

This chapter described two main points. First is describing how Conjure fits into the big picture: where does it fit as a part of a larger tool-chain. Second is the description of its internal design and architecture: the internal representation, parsing, type checking, the partial evaluator, and the two kinds of modelling transformations and how they are applied.

In combination with a description of the rule language, which is the subject of the next chapter, these two chapters describe the core ideas in Conjure.

Chapter 5

The Rule Language

This chapter explains the domain-specific rewrite rule language of Conjure. It starts by giving motivation for the existence of such a language in Section 5.1. Section 5.2 gives an overview of the different kinds of rules Conjure implements. The overall structure of each kind of rule is given in Section 5.3. In the rest of the chapter, features of the rule language are described. Section 5.4 describes a crucial part of the rule language, pattern matching. Section 5.5 defines many features and operators of the rule language. Section 5.6 discusses a system called bubbling that is used mostly when a rule needs to create auxiliary decision variables and Section 5.7 defines how unused names are generated during rule application and how such names can be requested in the definition of a rule.

5.1 Motivation for a rewrite rule language

Conjure converts problem specifications written in Essence into CP models written in Essence′. In order to do so, it needs to *represent* abstract decision variables and parameters of an Essence problem specification using concrete decision variables and parameters in an Essence′ model. Namely, Essence domains need to be transformed into Essence′ domains, and Essence expressions need to be transformed into Essence′ expressions.

The main duty of Conjure is applying transformations. It contains generic mechanisms to apply certain kinds of transformations to the AST representation of the problem

specification it is working on. There are benefits to separating the definition and application of transformations. One benefit is that Conjure knows exactly what each transformation can do and which parts of the AST it can modify. Another benefit is that transformations themselves do not have to implement a specific tree-walking mechanism, Conjure can reuse the generic tree walking mechanism for each transformation.

Conjure is implemented using the Haskell[Mar10] programming language. Therefore, the easiest way of defining transformations would be to use Haskell. In such a setting, transformations would merely be Haskell functions, accepting candidate expression fragments as arguments and producing replacement expressions if a transformation is applicable. This way, rules would directly be written in terms of the internal representation, the AST, of Essence. There are both advantages and disadvantages of using Haskell to implement transformations in Conjure. On the plus side: we would be reusing a lot of existing infrastructure from the Haskell programming language, application of transformations would be easier since they would be simple Haskell functions written in terms of the in-memory representation. However, there are also disadvantages of using Haskell. Firstly, rules will have to be defined in terms of the in-memory representation. This is an advantage when applying rules, but a disadvantage when defining them because it creates a very tight and fragile coupling between how the internal data structures are defined and rule definitions. It also creates a higher barrier to entry, one needs to be familiar with Haskell and the actual implementation of Conjure to be able to write new transformations. Furthermore, and maybe most importantly, definitions of transformations quickly become very verbose when a general purpose programming language is used.

Transformations in Conjure are implemented as rewrite rules. These rewrite rules are written in a domain-specific rewrite rule language, and they use the same syntax as Essence and Essence′ to a large extent. The rewrite rule language adds a number of new constructs; these are described in Section 5.5.

There are many advantages of using a domain-specific rewrite rule language to encode modelling transformations. The rule language only contains those features that we need for

the purpose and nothing more. Thanks to this limited set of language constructs and its resemblance to Essence, we hope using the rewrite rule will have less cognitive overhead. The rule language also makes it easier to extend Conjure with new rules. Adding a new rule does not require recompiling Conjure and it does not require writing any any Haskell code. Probably most importantly, rules written in the rule language are decoupled from the internal representation of Essence and Essence′. This makes it easier to maintain rules, changes to the internal data structures of Conjure will require no change to rules themselves.

Using a domain-specific rewrite rule language also has a few disadvantages. Firstly, the rules are interpreted rather than compiled. Compiled rules are likely to be faster; yet we chose to use a rule language because conciseness and ease of extensibility of rules were more important to us. Secondly, the rule language is not Turing-complete. Some transformations are harder, if not impossible, to write in the rule language in comparison to a Turing-complete programming language. This disadvantage does not seem to be a huge problem in practice, since we can always extend the rewrite rule language with new language constructs.

It is also important to note that having a domain-specific rewrite rule language does not mean Haskell cannot be used to define any transformations. Some transformations can be defined using Haskell, especially when defining the same transformation using the rule language is cumbersome or too slow. In practice, Conjure's partial evaluator is implemented using transformations defined in Haskell because of evaluation of Essence expressions happens very often and is desired to be fast.

5.2 Kinds of rules

Conjure contains two main kinds of rules: *representation selection* and *expression refinement* rules. These two kinds of rules directly capture two kinds of modelling decisions human modellers often make when they model a problem using CP. Broadly speaking, represent-

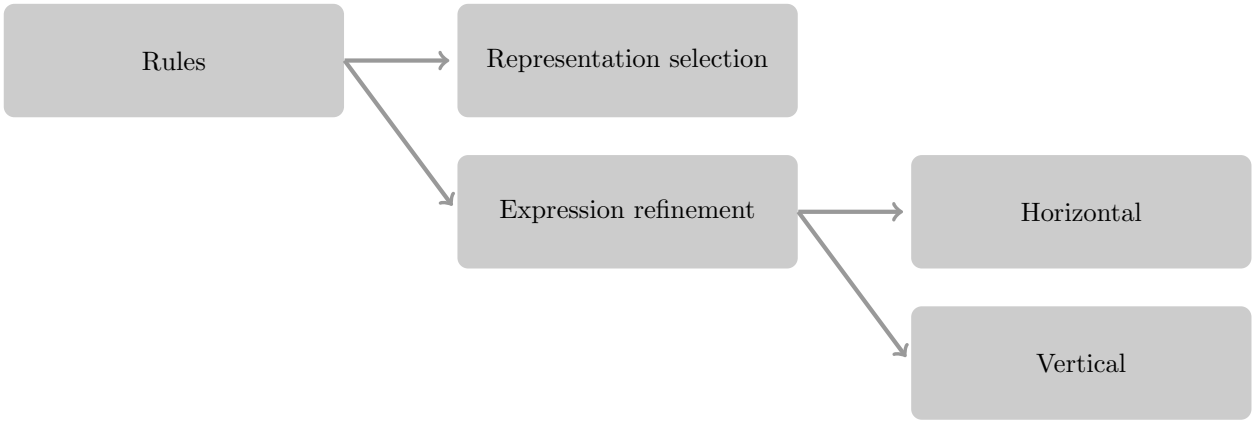


Figure 5.1: Hierarchy of different kinds of rules in Conjure.

ation selection rules capture the notion of viewpoint selection. They are used to encode concrete representation options for abstract decision variables of Essence. Expression refinement rules capture the notion of stating problem constraints using the selected viewpoint. Each constraint in the problem specification has to be stated in a different way depending on the selected viewpoint. Also note that a single constraint in Essence can be modelled in multiple different ways even after the viewpoints of all decision variables are fixed.

Representation selection rules take as input a domain. If a rule is applicable, it produces 3 outputs. First output is a replacement domain to be used to *model* or *represent* the input domain. Second output is a structural constraint to be posted after rule application to maintain invariants of the input domain. Third output is a name for the representation. This name will be referred to later when applying expression refinement rules.

Expression refinement rules take as input an expression, and produce as output a replacement expression. They are used to *model* or *refine* constraint expression and the objective expression. Essence constraints can be seen as abstract specification rather than actual constraints. They are refined to actual constraints, once viewpoints are chosen for each decision variable involved in the abstract specification of the constraint.

Expression refinement rules are further divided into two subcategories: *horizontal* and *vertical* expression refinement rules. Horizontal rules are used mostly to provide sensible

default refinements to Essence expressions in terms of other Essence expressions. They do not change the level of abstractness of an expression, i.e., they never change representations of any of the decision variables involved in the expression. On the other hand, vertical rules do change the representations of decision variables involved in expressions. They typically check the name of the representation for a decision variable and produce refinements depending on the representation selection.

An important point about expression refinement rules is the level of granularity they operate at. They do not transform whole constraints, rather they can transform any expression fragment. For example, an Essence expression like `a union b` can be refined independent of the constraint it occurs in.

The complete hierarchy of the different kinds of rules in Conjure is given in Figure 5.1.

5.3 Structure of a rule

5.3.1 Representation selection rules

A representation selection rule works on a domain and produces a concrete representation option for a decision variable or problem parameter with that domain.

Representation selection rules are composed of a preamble and a number of cases. The preamble contains the name of the rule, the output domain, and other common parts of the cases if there are any. Each case contains a *domain pattern*, optionally structural constraints, local guards, and local name bindings. Cases are separated by the `***` character sequence.

When applying a rule, each case is tried from top to bottom and the first matching case is applied. If one of the cases is applicable the whole rule is said to be applicable; and if none of the cases is applicable the whole rule is not applicable.

The syntax for each case follows the syntax given in Figure 5.2. The `domain-pattern` is used for pattern matching on the input domain. `letting` statements are used to introduce local bindings, and `where` statements are used to introduce guards. A rule is only applicable if all its guards can be reduced to `true`.

```
*** domain-pattern
    ~> structural-constraints
    where    ...
    letting  ...
```

Figure 5.2: The syntax for a case of a representation selection rule.

```
~> representation-name
~> replacement-domain
~> structural-constraints
    where    ...
    letting  ...
```

Figure 5.3: The syntax for a complete representation selection rule.

The syntax for a complete representation selection rule is given in Figure 5.3. The `where` and `letting` statements that are in the preamble are shared amongst all cases. They are mostly provided to reduce repetition. The structural constraints that are in the preamble are also posted in addition to the structural constraints that are posted by a case.

5.3.2 Expression refinement rules

The syntax for an expression refinement rule is given in Figure 5.4 Expression refinement rules replace expression by other expressions, and their syntax follows this. The left-hand side of the ~> sign is an expression-pattern. The input expression is pattern matched against this pattern (see. Section 5.4). `where` statements are checked in a similar fashion to those in representation selection rules. Each `where` statement needs to be reduced to the value `true` for a rule to be applicable. `letting` statements are used to introduce local bindings. `find` statements are specific to expression refinement rules, they are used to introduce auxiliary decision variables to the model. Details about how this mechanism works are given in Section 5.7 and Section 5.6.

Expression refinement rules optionally carry information about their *precedence level*. Although Conjure is designed to explore different models by applying all applicable rules, the rule precedence mechanism can be used to give Conjure an indication as to which rules

```
[precedence level]
expression-pattern ~> replacement-expression
    where    ...
    letting  ...
    find     ...
```

Figure 5.4: The syntax for an expression refinement rule.

to favour. Conjure will choose to apply rules at a lower precedence level in cases when multiple rules are applicable to a single expression fragment. Precedence levels are intended to be used when a specific rule is considered to be dominating a more general rule.

5.4 Pattern matching

The rule language makes extensive use of *pattern matching*. This section describes pattern matching as used by the rule language of Conjure.

Pattern matching is defined on two arguments: a *pattern* argument and an actual argument. A pattern is an Essence domain or expression, additionally equipped with meta-variables. There are two kinds of patterns, a domain pattern and an expression pattern. These do not need to be identified manually, Conjure will infer the kind of a pattern depending on context.

Pattern matching takes a pattern and an argument. It can be used to write conditionals depending on the *shape* of the argument with respect to the pattern. It can also be used to pull out fragments of the argument and bind them to meta-variables, ready for later use.

5.4.1 Meta-variables

The rule language shares and extends the Abstract Syntax Tree (see Section 4.2.1) of Essence. In particular, it adds meta-variables. Meta-variables are part of the rewrite rule language and they stand for elements of the object language, Essence. The rule language uses an ampersand symbol as a one letter prefix to any identifier to indicate a meta-variable.

In a pattern, meta-variables are used to denote parts of a domain or an expression. If pattern matching succeeds, meta-variables will be instantiated by Conjure to corresponding parts of the actual argument of pattern matching.

Once a pattern match succeeds and a value is assigned for each meta-variable, the meta-variables can then be used elsewhere in a rule. Conjure will replace any meta-variable with its value assigned by pattern-matching when applying a rule.

5.4.2 Semantics of pattern matching

Pattern matching in Conjure works by comparing AST representations of the pattern and the actual argument. The simplest pattern is a meta-variable by itself, and this pattern would match any actual argument. A compound pattern is represented by an Essence AST. In the case of compound patterns, the shape of the pattern tree and the shape of the actual argument tree are compared. If they both have the same node type and same number of children, both trees are said to have the same shape at this level. Pattern matching proceeds to recursively check each child of the pattern to the corresponding child of the argument.

5.4.3 An alternative

An alternative to pattern matching in Conjure's rule language would be to provide a collection of operators which check the shape of an Essence domain or expression. These operators would then be used in the `where` statements to conditionally apply rules. Moreover, the rule language would also need to provide a collection of operators to pull out fragments of an Essence domain or expression. These operators would then be used in the local `letting` statements to assign values to meta-variables.

Such an alternative would have been more cumbersome, would require the addition of a large number of operators to the rule language, and would slow down rule application. Pattern matching using a syntax very close to the syntax of Essence is chosen instead.

Algorithm 5.1: Pattern matching in Conjure's rule language

```
Algorithm patternMatch(pattern, argument)
    switch pattern do
        case MetaVariable(name)
            bind(name, argument)
        end
        case Tree(label1,children1)
            switch argument do
                case Tree(label2,children2)
                    if sameLabel(label1,label2) AND sameLength(children1,children2) then
                        for (p,a) ← zip children1 children2 do
                            patternMatch(p,a)
                        end
                    else
                        reportError(pattern,argument)
                    end
                end
                otherwise
                    reportError(pattern,argument)
                end
            endsw
        end
        otherwise
            reportError(pattern,argument)
        end
    endsw
```

5.5 Rule language constructs

The rule language provides a small number of language constructs on top of Essence. The additions are mostly in the form of new operators. These operators have special meaning and are handled specifically by Conjure. `where`, `letting`, and `find` statements are taken from Essence and given new meanings.

5.5.1 Guarded rewriting: `where` statements

Essence uses `where` statements for validity checking of instance data. The rule language reuses `where` statements as syntax for guards. Guards are used in most rewriting system to limit the applicability of a rule. A rule is only applied if its guards can be evaluated to `true`.

5.5.2 Guarding operators: `has*`

These operators are commonly used in the `where` statements of a rule.

All three operators listed in this section evaluate to a boolean value. They can be freely used in conjunction with existing Essence operators. For example, a `where` statement in a rule can contain a disjunction (`\/`) of two expressions both built using the rule language operator `hasRepresentation`.

5.5.2.1 `hasDomain`

The operator `hasDomain` takes two arguments. Its first argument is an expression. Its second argument is a *domain pattern*. The second argument gives the expected domain for the first argument. `hasDomain` is used in infix form.

An expression of the form `&x hasDomain ` set (size &n) of &tau` will be evaluated to `true` if the expression bound to the meta-variable `&x` has a domain that matches the pattern on the right. In order to match this pattern, the domain of `&x` has to be a set domain, with a known cardinality. It also needs to not have any other attributes. Domain patterns are described in more detail in Section 5.4. In addition to evaluating to `true`, such an expression will also introduce values for meta-variables `&n` and `&tau`. `&n` will be bound to an integer expression, whereas `&tau` will be bound to a domain. The same expression will be evaluated to `false` if the domain of `&x` cannot be successfully pattern matched to the domain on the right-hand side. In this case, the meta-variables `&x` and `&tau` will not be bound to any value.

5.5.2.2 `hasType`

The operator `hasType` is similar to `hasDomain`. It also takes two arguments, and is used in infix form. Its first argument is an expression, its second argument is a *type pattern*. In Essence, every expression has a type. The type of an expression can be calculated by deleting annotations from its domain.

For example, if a decision variable has the domain `set (size 3) of int(0..9)`, its type will be `set of int`.

5.5.2.3 `hasRepresentation`

The operator `hasRepresentation` is only used in vertical expression refinement rules. Its first argument is an expression, and its second argument is an identifier.

An expression of the form `&x hasRepresentation Matrix1D` will be evaluated to `true` if `&x` is bound to an abstract decision variable or problem parameter that is being represented using the `Matrix1D` representation. It will be evaluated to false otherwise.

5.5.3 `refn`

Depending on context, the operator `refn` can be seen as a shorthand for *'refinement'* or *'refinement of'*.

In a representation selection rule, `refn` does not take any arguments. It is merely an identifier which is handled specifically during rule application. Representation selection rules are written in terms of domains and not full declarations of decision variables or problem parameters. When applying a rule, Conjure projects the domain out of a declaration statement and uses it for pattern matching. Representation selection rules also do not output full declarations by themselves. They output a replacement domain instead, and Conjure is responsible to generate a new declaration together with a new name using the generated domain.

This arrangement limits the ability to refer to the generated decision variable within the representation selection rule itself. Because the new decision variable is not even give a name yet. The operator `refn` is used to fill this gap, it can be used in a representation selection rule to refer to the decision variable which will be generated as a result of applying the rule. For example, if a rule outputs a matrix domain and the rule wants to post an `allDiff` constraint on this matrix the syntax to use is `allDiff(refn)`.

The same `refn` keyword is used in expression refinement rule to have a very similar meaning. There is one important difference though: in expression refinement rules, `refn` is actually an operator which takes a single argument. It takes as argument an abstract decision variable or problem parameter and returns the refinement of the argument. In order to do this, the argument has to have a representation selected for it in the representations selection phase. Typically, a rule will use a `hasRepresentation` operator in a `where` clause to check which representation something has, before calling `refn` on it.

5.5.4 `domSize`

The operator `domSize` is a shorthand for *'domain size'*. It takes as input an Essence expression and returns an integer expression representing the size of the domain of its argument. For constant expressions, it returns `1`.

For example, `domSize(int(1..9))` is 9, `domSize(set (size 2) of int(1..9))` is `9 ** 2` which is equal to `81`.

5.5.5 Deep replace

In rules, sometimes all occurrences of a certain subexpression needs to be replaced by some other subexpression. Conjure provides a way to accomplish this in the rule language: the deep-replace operator.

The syntax for deep-replace is demonstrated by example: `&body  i -> m[i]` . In this example, `&body` is a meta-variable, `i` and `m` can be any expression. The effect of using the deep-replace operator is replacing all occurrences of `i` in `&body` with `m[i]`.

For example if `&body` is `(i + 3) ** j`, after deep-replace if the result will be `(m[i] + 3) ** j`.

5.6 Locality of rules and 'Bubbling'

In Essence, constraints are top level boolean expressions. Top level boolean expressions are very nice to work with, attaching additional conditions to an expression as part of a rewrite is as simple as using a conjunction operator and introducing auxiliary variables is as simple as adding top level decision variables to the original model. Boolean expressions nested inside other boolean expressions are a little bit harder to work with. Posting additional conditions during a rewrite is equally easy to, the conjunction operator can still be used. However, when introducing auxiliary variables special machinery is needed. When the boolean expression is nested inside other expressions but is not inside a quantified context the auxiliary variable can freely be added to the top level. When the boolean expression is nested inside a quantified context, an array of auxiliary variables needs to be created for each iteration of the quantified expression. Furthermore, the trickiest case is working with non-boolean expressions. For non-boolean expressions, posting additional conditions cannot be achieved using the conjunction operator. Instead, the condition needs to be posted to the closest boolean context.

Bubbling is a technique developed to overcome this specific problem. Using bubbling, rule authors do not need to think about the type or nesting of the input expression for a rule. A rule indicates additional conditions and decision variables by putting them inside a *bubble*. Conjure automatically handles the transfer of the constraints and decision variables in a bubble.

The definition of rules in Conjure is very *local*. An expression refinement rule does not have access to parent of an expression. This means, an expression refinement rule can do one thing: pattern match on an expression fragment and replace it with another. Representation selection rules are not much different either: they can only match the domain of a top level decision variable and produce a replacement. (In addition, representation selection rules can also post top level constraints in the form of *structural constraints*.)

This simplification is helpful because it simplifies the implementation of rule application.

Moreover it makes reasoning about rule application simpler.

On the other hand, this restriction also limits the abilities of the rule language. There are many valid reasons for a rule to introduce new decision variables, or post additional constraints outside the scope of the expression fragment it is working on.

Posting additional constraints is as easy as producing a conjunction of the actual and the additional constraint for boolean expressions. For any other expression type, it is not so easy.

The prototype implementation discussed in [Fri+05b] operated by matching against and rewriting *complete constraints* after flattening all expressions by introducing auxiliary variables and further constraints. However, such an approach has a number of drawbacks. It may not be scalable in general as we may possibly have a huge number of constraint types that look very similar but slightly different (such as: `x subseteq (a union b)` and `x supseteq (a union b)`). Furthermore, a large number of rewrite rules may be needed, one for each constraint type. Finally, the flattening process may introduce a large number of unnecessary auxiliary variables and changes the structure of our constraints for which we may have better rewrite rules that exploit that structure.

Using an approach called *bubbling*, we overcome these drawbacks by allowing the rules to match and rewrite *expressions* within a constraint rather than (necessarily) the whole constraint. This allows us to accomplish three things. First, we can refine a greater proportion of the Essence language using fewer rules. Second, unlike the prototype in [Fri+05b], we no longer need to *flatten* a specification prior to refinement, avoiding introducing unnecessary auxiliary variables. Finally, we may have optimised rewrite rules for constraints with specific structure. For instance, consider the constraint `(a union b) subseteq c`. If we flatten it, we would have `x subseteq c /\ x = a union b` which introduces an auxiliary variable and requires refining unnecessarily a set equality constraint. Using the bubbling approach, in addition to this refinement, we may rewrite this into a conjunction of two `subseteq` constraints, namely `a subseteq c /\ b subseteq c`, by having a dedicated rewrite rule which reasons about the structure of this type of constraint.

There is a subtle problem arising when we match an expression fragment and rewrite it to an equivalent expression fragment. The rewrite might introduce extra constraints and auxiliary variables.

For instance, consider the following Essence specification:

```
given lb,ub,n,m,k : int
find t : set (size n) of int(lb..ub)
find A : set (size n) set (size m) of int(lb..ub)
such that
  forAll s in A . (max(s) - max(t) = k) -> k in s
```

The rewrite rule for the set max operator (`max(s)`) needs to introduce an auxiliary variable, say `max_s`, along with constraints that enforce that `max_s` is the maximum element in set `s`. We refer to these extra constraints as *helper constraints*.

We equip our rewrite rules with an extra operator "`{ x @ b }`" which attaches a "bubble" `b` containing declaration of auxiliary variables and helper constraints any expression, in this case `x`. For example, our rewrite rule for the set max operator is as follows:

```
max(&s) ~> { aux
             @ find aux : &tau
               such that
                 forAll i in &s . i <= aux,
                 aux in &s
             }
  where &s hasDomain `set (..) of &tau`
```

If we apply this rule of the above example which contains two set max operators, we end up with the following resulting expression:

```
forAll s in A .
    ({max_s @ bubble_s} - max(t) = k) -> k in s
```

```
1 forAll s in A .
2     ({max_s @ bubble_s} - {max_t @ bubble_t} = k) -> k in s
```

As we can see, the intermediate expression is not a valid Essence expression one yet. In fact, we need to move the bubbles to their correct positions. Conjure contains built-in rules to move constraints in bubbles upwards until they reach a boolean context and then attach the constraints using a conjunction to the closest boolean context. The process of moving constraints upwards is described below. Auxiliary decision variable declarations that are in bubbles are given a unique name automatically by Conjure and introduced at the top level. Also if the bubble happens to be in the body of a quantified expression, a matrix of auxiliary variables is created automatically, and the constraints in the bubble are automatically lifted to work on a the corresponding index of the matrix.

Bubbling up happens one level at a time. If a bubble is attached to a boolean expression, it is turned into a conjunction and left in place. Otherwise, for an expression `e` if the immediate children of `e` have bubbles attached to them, the constraints in the bubble are collected and attached to `e`. This operation is performed recursively from the bottom-up until all constraints in bubbles are converted into conjunctions.

In our running example, these are the results of successively applying the bubbling-up operation.

```
1 forAll s in A .
2     ({(max_s - max_t) @ (bubble_s /\ bubble_t)}) = k) -> k in s
```

```
1 forAll s in A .
2     ({(max_s - max_t = k) @ (bubble_s /\ bubble_t)}) -> k in s
```

At this point, the bubbles are attached to a boolean expression and they can safely be converted to a conjunction "/\" resulting in the following valid Essence expression:

```
1 forAll s in A .
2     ({(max_s - max_t = k) /\ (bubble_s /\ bubble_t)}) -> k in s
```

Skolemisation as an alternative to Bubbling

Some transformation rules in Conjure are already local: they match a single expression fragment and replace it with another. A simple example of this is simplification rules like replacing '`true -> &a`' with '`&a`'. Other rules need to modify some global state, in the case of Conjure this happens in the form of the need to introduce an auxiliary decision variable. Bubbling is essentially an abstraction enabling rules to introduce new auxiliary decision variables and post constraints in non-relational constraints. These are put inside a bubble within the rule and are placed at the correct place by Conjure.

Skolemisation[Hod97] is commonly performed by automated theorem provers to remove existential quantifiers from logic statements and replace them with top level decision variables. The general issue of applying a skolemisation transformation to CP models is recently studied in the literature[Jef+10]. This paper shows that depending on the solver implementation using existential quantifiers can actually be better than introducing auxiliary variables. However, in general the introduction of auxiliary variables can improve constraint propagation[Smi06a].

The decision of whether to convert top level decision variables into existential quantification or turning existential quantification into top level decision variables depends on many factors, and this decision is orthogonal to the technique presented here. Bubbling can be viewed as a similar technique to skolemisation only because the output of both techniques is the creation of new decision variables.

5.7 Generating unused names

Transformations in Conjure need to be able to generate unused names during their application. However, they do not know about the whole context of rule application, so abstracting

this notion and providing functionality to produce unused names automatically will make the life of a rule author easier.

Rules need new name generation in two places. First is quantified expressions, which contain a quantified variable whose name needs to be unique in this context. Second is when a rule introduces auxiliary decision variables using a `find` statement.

In both of these cases, the rule author needs to use a unique name in the context of the rule, and Conjure guarantees to replace it with a unique name in the context of the model at the time of rule application.

5.8 Summary

This section defined the domain specific rewrite rule language used in Conjure. It gives motivation for the existence of such a language, defines the kinds of rules Conjure has, and describes the features of the rule language. The next section will give a selection of rules to demonstrate how the complete system is put together.

Chapter 6

The Rules of Conjure

This chapter describes the rules of Conjure. It starts describing the mechanism used to test the correctness of Conjure's rules, followed by a listing of the representation selection rules and vertical rules for each representation and for each abstract type constructor in Essence, and horizontal rules which are representation independent.

The rules of Conjure are expected to be free of infinite loops; however, this condition is not checked by Conjure. Provided with a collection of rules which go into an infinite loop for certain inputs, Conjure will go into an infinite loop of rule applications and will never halt. The rules do not need to be confluent. Indeed, different rule application orders are explored by Conjure to produce different outputs models.

In this thesis, the correctness of the provided rules is not formally proven. In order to provide such a proof, firstly each individual rule has to be proven correct. Secondly, the complete collection of rules has to be proven correct by considering the interaction between different variable representations and expression refinement rules. The implementation of Conjure together with the rules presented in this chapter are tested thoroughly for correctness.

For testing Conjure, a collection of 1200 problem specifications written in Essence is used. More than half of these specifications are written to test different parts of Conjure and its rule base during the implementation of Conjure and during the addition of new

variable representations. Essence problem specifications for all problems in the online Essence Specification Catalog[1] are also used for testing. Most of the problems in the Essence Catalog are taken from CSPLib[2] which is a frequently used library of test problems for CP.

Each problem specification is run through Conjure to generate all attainable models using the collection of rules presented in this chapter. For each problem specification, a collection of parameter files are used to instantiate each generated model, and then the problem instance is solved using SavileRow and Minion as presented in Section 4.1. Each solution is then validated using Conjure.

Conjure takes the original input problem specification, the input parameter and the generated solution for its solution validation. For validation, Conjure instantiates the `given` statements using the values present in the input parameter and instantiates the `find` statements using the values present in the solution file. At this point, the built-in Essence evaluator in Conjure is able to reduce the input problem specification down to an atomic boolean value: `false` or `true`. A `false` value indicates an invalid solution and a `true` value indicates a valid solution.

This approach has a good power-to-weight ratio for testing the correctness of a system as complex as Conjure. All the moving parts of the system are tested since we start from as high level as Essence problem specifications and data presented using Essence data structures, go all the way down to a concrete solver and translate the solutions back up to the same abstraction level we started from.

The rest of this chapter presents the representation selection rules and vertical rules for each representation and for each abstract type constructor in Essence; and horizontal rules which are representation independent.

Sets are explained first in Section 6.1. Each set representation is listed as a subsection, starting with the representation selection rule, followed by the vertical rules required for that representation. After all the representations are listed, some horizontal rules used for

[1] http://www.cs.york.ac.uk/aig/constraints/AutoModel/Essence/specs120
[2] http://www.csplib.org

```
1  ~> Set~Occurrence
2  ~> matrix indexed by [&tau] of bool
3    where &tau hasType 'int'
4
5  *** set of &tau
6
7  *** set (size &size_, ..) of &tau
8    ~> (sum i : &tau . toInt(refn[i])) = &size_
9
10 *** set (minSize &minSize_) of &tau
11   ~> (sum i : &tau . toInt(refn[i])) >= &minSize_
12
13 *** set (maxSize &maxSize_) of &tau
14   ~> (sum i : &tau . toInt(refn[i])) <= &maxSize_
15
16 *** set (minSize &minSize_, maxSize &maxSize_) of &tau
17   ~> (sum i : &tau . toInt(refn[i])) >= &minSize_
18   /\  (sum i : &tau . toInt(refn[i])) <= &maxSize_
```

Figure 6.1: Occurrence representation for set domains.

the representation are also given before finishing the section. The same is done for multi-sets in Section 6.2, for functions in Section 6.3, for relations in Section 6.4, and for partitions in Section 6.5. Section 6.6 gives a listing of some of the most important horizontal rules in Conjure to provide the reader with a general understanding of the mechanism.

6.1 Rules for set domains

6.1.1 Occurrence representation

Occurrence representation for sets works on sets of integers. It uses a matrix of booleans indexed by the possible elements of the set. Membership is denoted by a `true` assignment to the corresponding position in the matrix. i'th position of the matrix is `true` iff i is a member of the set, and `false` if it is not. Set cardinality is not stored separately as it can easily be calculated using a `sum` over all the booleans. One advantage of the occurrence representation is its uniform applicability to integer sets independent of the size attributes of a set. It can

be applied to both those sets with a known cardinality and those with a variable cardinality.

Figure 6.1 gives the rule used by Conjure to implement representation selection for the Occurrence representation. The first line gives a name to the representation. The second line gives the output domain: when applied, this rule always generates a domain of the form `matrix indexed by [&tau] of bool`. In this domain, `&tau` is a meta-variable and its value is not known yet. Line 3 is a condition: this rule is only applicable if `&tau` has an integer type. After this preamble 5 cases are listed. Each case has a domain pattern and optionally structural constraints.

The first case contains a set domain without any size attributes. In such a case no structural constraints can be posted: any assignment to the matrix domain gives a valid assignment to the set domain. The second case contains a set domain with a `size` attribute. The pattern also contains a `'..'` next to the `size` attribute, this syntax indicates that the pattern should ignore other attributes of this domain if there are any. The `'..'` syntax can be used in this case because once a `size` attribute is given, other attributes of a set domain are irrelevant. The third and forth cases contain set domains with a `minSize` and `maxSize` attribute respectively. They post appropriate structural constraints to constrain the cardinality of the set. The fifth and the final case contains a set domain with both the `minSize` and the `maxSize` attributes. In this case, a conjunction of two constraints are posted to constrain the cardinality of the set from both ends.

6.1.1.1 Vertical rules

The rule given in Figure 6.2 is used when refining a quantified expression over set decision variables or parameters that are represented using the Occurrence representation. It matches all three kinds of quantified expressions in Essence: `forAll`, `exists` and `sum`. The quantifier keyword is bound to the meta variable `&quan`. It replaces a quantified expression over a set decision variable into a simple quantified expression, one quantifying over an integer domain. The quantified variable `&i` represents elements in the set in the original expression, and it represents indices of the matrix in the output expression. Indices of a set with the

```
1  [1000]
2
3  &quan &i in &s , &g . &k
4      ~>
5  &quan &i : &tau , &g /\ &m[&i]
6                  . &k
7
8    where &s hasDomain `set (..) of &tau`
9    where &s hasRepr Set~Occurrence
10
11   letting &m be refn(&s)
```

Figure 6.2: Vertical rule for Quantified expressions and Occurrence representation of sets

```
1  [900]
2
3  &x in &s ~> refn(&s)[&x] = true
4
5      where &s hasRepr Set~Occurrence
```

Figure 6.3: Vertical rule for membership operator and Occurrence representation of sets

occurrence representation correspond to elements of the set if the matrix contains a `true` value in the corresponding position. A new guard is added to the output expression so the *body* of the quantified expression is only active for those values that are in the set.

This vertical rule is the only rule required for representations of set domains. All other set operators can be refined using horizontal rules. These horizontal rules are given in Section 6.6.1.

For example the expression « `forAll i in a , i > 3 . i in b` » will be refined to « `forAll i : int(..) , i > 3 /\ a'[i] . i in b` » using this rule, where `a'` is the refinement of a.

Figure 6.3 defines a vertical rule which is not necessary for completeness. If this rule was left out, Conjure would use a horizontal rule (Figure 6.37) to rewrite expressions using the `in` operator into expressions using an `exists` quantified expression to be further refined using the existing vertical rule for quantified expressions: Figure 6.2.

```
1 ~> Set~Explicit
2 ~> matrix indexed by [int(1..&size_)] of &tau
3 ~> allDiff(refn)
4
5 *** set (size &size_, ..) of &tau
```

Figure 6.4: Explicit representation for set domains with fixed cardinality.

For example the expression « `3 in s` » will be refined to « `s'[3]` » using this rule, where `s'` is the refinement of s.

6.1.2 Explicit representation with a fixed cardinality

The simple version of the Explicit representation for sets works on sets with fixed cardinality. Its implementation is very simple, yet it is very widely applicable since the rule does not place a condition on the type of `&tau`, the inner type of the set domain.

Figure 6.4 gives the rule used by Conjure to implement representation selection for this representation. The rule contains only one case: a set domain with a fixed `size`. Its output domain is a matrix with as many elements as the set requires. The most important component of this rule is the structural constraint.

A set has the implicit condition that each element contained in the set needs to be distinct. Using a matrix of elements to represent a set requires the addition of an explicit condition, in the form of a structural constraint, to maintain this requirement of the original domain. For this purpose, an `allDiff` constraint is posted on the matrix. This constraint will need to be decomposed into a clique of inequality constraints if `&tau` is not a compatible type with the `allDiff` constraint supported by Essence′. The decomposition of `allDiff` is implemented using the horizontal rule shown in Figure 6.55.

6.1.2.1 Symmetry

This representation introduces *modelling symmetry*: items in the set can be ordered. It is not easy to write this ordering constraint instead of the `allDiff` here because `&tau` can be any

```
1 [1000]
2
3 &quan &i in &s , &g . &k
4   ~>
5 &quan j : &r , &g { &i --> &m[j] }
6                   . &k { &i --> &m[j] }
7   where &s hasRepr Set~Explicit
8   letting &m be refn(&s)
9   letting &r be indices(&m,0)
```

Figure 6.5: Vertical rule for Quantified expressions and Explicit representation of sets

type and Essence does not contain a generic operator to order values of arbitrary types. However, Conjure can automatically break this kind of symmetry. The mechanism will be described in Chapter 8.

6.1.2.2 Vertical rules

The rule given in Figure 6.5 is used when refining quantified expressions over set decision variables which are represented using the Explicit representation. In this rule, the quantified variable `&i` in the input expression takes values from the elements of the set. However, `j` in the output quantified expression takes values from the indices of the matrix. The guard `&g` and the body `&b` of the quantified expression are written in terns of the elements of the set, and the elements of the set are represented using items in the Explicit matrix. For this reason the *deep-replace* construct is used to replace all references to `&i` with the corresponding expression `&m[j]`.

For example the expression « `forAll i in a , i > 3 . i in b` » will be refined to « `forAll j : int(..) , a'[j] > 3 . a'[j] in b` » using this rule, where `a'` is the refinement of `a`.

6.1.3 Explicit representation with variable cardinality and Boolean markers

The simple Explicit representation only works for sets with known cardinality. This is a huge limitation in practice, because the cardinality is also a decision rather than a parameter

```
1  ~> Set~ExplicitVarSize~BoolMarker
2  ~> matrix indexed by [int(1..&maxSize_)] of (bool,&tau)
3  ~> forAll i,j : int(1..&maxSize_)
4        , i < j /\ refn[i][1] /\ j <= refn[j][1]
5        . refn[i][2] != refn[j][2]
6
7  *** set of &tau
8    letting &maxSize_ be domSize(&tau)
9
10 *** set (minSize &minSize_) of &tau
11   ~> (sum i : int(1..&maxSize_) . toInt(refn[i][1])) >= &minSize_
12   letting &maxSize_ be domSize(&tau)
13
14 *** set (maxSize &maxSize_) of &tau
15
16 *** set (minSize &minSize_, maxSize &maxSize_) of &tau
17   ~> (sum i : int(1..&maxSize_) . toInt(refn[i][1])) >= &minSize_
```

Figure 6.6: Explicit representation with variable cardinality and Boolean markers

in many interesting problems which use a decision variable with a set domain. This representation works for sets of all types and does not require a fixed cardinality.

There are four cases in the representation selection rule given in Figure 6.6. The first two cases do not have a `maxSize` attribute, and hence the maximum number of elements that the set domain can have needs to be calculated. This calculation is done using the `domSize` operator. The value is bound to a local meta variable `&maxSize_` to be used in the preamble. The last two cases do not need to use the `domSize` operator as they already have access to the attribute value provided in the domain.

The domain generated by this representation uses a matrix which has enough slots for the maximum number of elements that can be in the set domain. Each item in the matrix is a 2-tuple; the first component is a boolean marker indicating whether the value of the second component should be treated as a member of the set. The structural constraint posted by this representation is similar to a decomposition of a conditional `allDiff` constraint.

In addition to the structural constraint posted in the preamble of the rule, cases 2 and 4 which have access to a `&minSize_` attribute post an additional structural constraint

```
1  [1000]
2
3  &quan &i in &s , &g . &k
4    ~>
5  &quan j : &r , &g { &i --> &m[j][2] } /\ &m[j][1]
6                . &k { &i --> &m[j][2] }
7
8    where &s hasRepr Set~ExplicitVarSize~BoolMarker
9    letting &m be refn(&s)
10   letting &r be indices(&m,0)
```

Figure 6.7: Vertical rule for Quantified expressions and Explicit-BoolMarker representation of sets

6.1.3.1 Symmetry

This representation introduces modelling symmetry in two places. First, the boolean markers can be ordered so that the `true` ones are at the beginning (or at the end) of the matrix. Second, the second components of each matrix element can be ordered provided the corresponding boolean marker is `true`.

6.1.3.2 Vertical rules

Figure 6.7 gives the rule for refining quantified expressions over sets with this representation. It works similarly to other vertical rules, the input expression is quantified over the set variable but the output expression is quantified over the indices of the matrix representation. Since the membership condition is captured using the boolean component in this representation, that component is used as a guard in the output quantified expression.

For example the expression « `forAll i in a , i > 3 . i in b` » will be refined to « `forAll j : int(..) , a'[j,2] > 3 /\ a'[j,1] . a'[j,2] in b` » using this rule, where `a'` is the refinement of `a`.

```
1  ~> Set~ExplicitVarSize~IntMarker
2  ~> ( int(0..&maxSize_)
3       , matrix indexed by [int(1..&maxSize_)] of &tau
4       )
5  ~> forAll i,j : int(1..&maxSize_)
6        , i < j /\ i <= refn[1] /\ j <= refn[1]
7        . refn[2][i] != refn[2][j]
8
9  *** set of &tau
10   letting &maxSize_ be domSize(&tau)
11
12 *** set (minSize &minSize_) of &tau
13   ~> refn[1] >= &minSize_
14   letting &maxSize_ be domSize(&tau)
15
16 *** set (maxSize &maxSize_) of &tau
17
18 *** set (minSize &minSize_ , maxSize &maxSize_) of &tau
19   ~> refn[1] >= &minSize_
```

Figure 6.8: Explicit representation with variable cardinality and an integer marker

6.1.4 Explicit representation with variable cardinality and an integer marker

This representation is very similar to Figure 6.6. Instead of using a boolean for each candidate member of the set, it uses a single integer decision variable. This integer represents the cardinality of the set and indices less than or equal to the value of it are considered to be members of the set. The output domain and the structural constraints are modified accordingly.

6.1.4.1 Symmetry

This representation avoids introducing one of the two modelling symmetries introduced when boolean markers are used. We no longer need to order boolean markers. However the second kind of symmetry is still introduced: values in the matrix up to the index marked by the integer can be ordered.

```
1 [1000]
2
3 &quan &i in &s , &g . &k
4   ~>
5 &quan j : &r , &g { &i --> &m[2][j] } /\ j <= &m[1]
6                 . &k { &i --> &m[2][j] }
7   where &s hasRepr Set~ExplicitVarSize~IntMarker
8   letting &m be refn(&s)
9   letting &r be indices(&m[2],0)
```

Figure 6.9: Vertical rule for Quantified expressions and Explicit-IntMarker representation of sets

```
1 [600]
2
3 |&s| ~> &m[1]
4   where &s hasRepr Set~ExplicitVarSize~IntMarker
5   letting &m be refn(&s)
```

Figure 6.10: Vertical rule for cardinality and Explicit-IntMarker representation of sets

6.1.4.2 Vertical rules

Figure 6.9 gives the rule for refining quantified expressions over sets with this representation. The rule is very similar to Figure 6.7, the biggest difference is in the guarding mechanism. This rule uses the integer marker rather than the boolean marker, since the membership condition is captured using the integer component.

For example the expression « `forAll i in a , i > 3 . i in b` » will be refined to « `forAll j : int(..) , a'[2,j] > 3 /\ j <= a'[1] . a'[2,j] in b` » using this rule, where `a'` is the refinement of `a`.

Figure 6.10 gives another vertical rule for this representation. Since this representation represents the cardinality of the set in a decision variable already, the cardinality operator can be implemented in terms of that decision variable. This rule overrides Figure 6.33 and Figure 6.34 because it is defined at a lower level (600).

```
1  ~> Set~ExplicitVarSize~Dummy
2  ~> matrix indexed by [int(1..&maxSize_)] of int(&lb..&dummy)
3  ~> forAll i,j : int(1..&maxSize_)
4        , i < j /\ refn[i] != &dummy /\ refn[j] != &dummy
5        . refn[i] != refn[j]
6
7    where &tau hasType 'int'
8    where &tau hasDomain 'int(&lb..&ub)'
9    letting &dummy be &ub + 1
10
11 *** set of &tau
12   letting &maxSize_ be domSize(&tau)
13
14 *** set (minSize &minSize_) of &tau
15   ~> (sum i : int(1..&maxSize_) . toInt(refn[i] != &dummy))
16       >= &minSize_
17   letting &maxSize_ be domSize(&tau)
18
19 *** set (maxSize &maxSize_) of &tau
20
21 *** set (minSize &minSize_, maxSize &maxSize_) of &tau
22   ~> (sum i : int(1..&maxSize_) . toInt(refn[i] != &dummy))
23       >= &minSize_
```

Figure 6.11: Explicit representation with variable cardinality and a dummy value

6.1.5 Explicit representation with variable cardinality and a dummy value

This representation is specialised to domains of type `set of int`. It works by introducing a dummy value to the domain and only considering values as a member of the set when their value is different from the dummy value. The dummy value is chosen by incrementing the largest integer in the original integer domain.

This rule is applicable for a smaller number of domains than those listed in the previous sections; however, it is likely to produce better models when it is applicable.

6.1.5.1 Symmetry

This representation introduces modelling symmetry. Values of the matrix that are different from the dummy value can be ordered. Moreover, those cells which hold the dummy value

```
1  [1000]
2
3  &quan &i in &s , &g . &k
4    ~>
5  &quan j : &r , &g { &i --> &m[j] } /\ &m[j] != &dummy
6                . &k { &i --> &m[j] }
7    where &s hasDomain 'set (..) of int(&lb..&ub)'
8    where &s hasRepr   Set~ExplicitVarSizeWithdummyault
9    letting &m be refn(&s)
10   letting &r be indices(&m,0)
11   letting &dummy be &ub + 1
```

Figure 6.12: Vertical rule for Quantified expressions and Explicit-Dummy representation of sets

can also be shifted to either the beginning or the end of the matrix.

6.1.5.2 Vertical rules

Figure 6.12 gives the rule for refining quantified expressions over sets with this representation. In the input expression of this rule `&i` represents an element of the set, in the output expression `j` represents an index in the representation matrix. For this reason the *deep-replace* construct is used to replace all references to `&i` with `&m[j]` in the guard and the body components of the quantified expression. Moreover, a new guard is added to post the condition of membership in the set: `m[j] != &dummy`.

For example the expression « `forAll i in a , i > 3 . i in b` » will be refined to « `forAll j : int(..) , a'[j] >3 /\ a'[j] != d . a'[j] in b` » using this rule, where `a'` is the refinement of `a` and `d` is the dummy value for `a`'s domain.

6.2 Rules for multi-set domains

6.2.1 Occurrence representation for Multi-Sets

The Occurrence representation for multi-sets can be used for domains with type `mset of int`. This representation is similar to the Occurrence representation for sets; however, it

```
1  ~> MSet~Occurrence
2  ~> matrix indexed by [&tau] of int(0..&maxOccur_)
3    where &tau hasType 'int'
4
5  *** mset (size &size_ , maxOccur &maxOccur_) of &tau
6  ~> ((sum i : &tau . refn[i]) = &size_)
7
8  *** mset (minSize &minSize_ , maxSize &maxSize_ ,
9            minOccur &minOccur_ , maxOccur &maxOccur_) of &tau
10 ~> ((sum i : &tau . refn[i]) >= &minSize_) /\
11     ((sum i : &tau . refn[i]) <= &maxSize_) /\
12     (forAll i : &tau , refn[i] > 0 . refn[i] >= &minOccur_)
13
14 *** mset (maxSize &maxSize_ , minOccur &minOccur_) of &tau
15 ~> ((sum i : &tau . refn[i]) <= &maxSize_) /\
16     (forAll i : &tau , refn[i] > 0 . refn[i] >= &minOccur_)
17   letting &maxOccur_ be &maxSize_
18
19 ...
```

Figure 6.13: Occurrence representation for Multi-Sets

uses integers instead of booleans in a matrix domain. The integer at each slot of the matrix indicates the number of occurrences of the index value in the multi-set.

In Essence, multi-sets have 5 attributes: `size`, `minSize`, `maxSize`, `minOccur`, `maxOccur`. The first three of these attributes control how many elements will be in a multi-set, and the last two control how many times each value can occur in a multi-set.

The representation rule given in Figure 6.13 is partial. The actual rule contains more cases to cover other combination of attributes. The three cases are chosen to exemplify the operation of the rule in combination with the preamble of the rule, which is listed in full.

In the first case, `size` and `maxOccur` attributes are given. The value of `&maxOccur_` is used in the output domain, and the value of `&size_` is used to post a structural constraint. In the second case, `size` is not given, however `minSize` and `maxSize` are given. The structural constraint changes to make sure the number of elements in the multi-set are between the given range. In addition, a `minOccur` attribute is also given. This means if a value is in the multi-set, it needs to be there are least `&minOccur_` times. A structural constraint to ensure

```
1 [1000]
2
3 forAll &i in &s , &g . &k
4   ~>
5 forAll j : &t , &g { &i --> j } /\ (&m[j] > 0)
6                 . &k { &i --> j }
7   where &s hasRepr MSet~Occurrence
8   where &s hasDomain `mset (..) of &t`
9   letting &m be refn(&s)
```

Figure 6.14: Vertical rule for `forAll` Quantified expressions and Occurrence representation of multi-sets

```
1 [1000]
2
3 sum &i in &s , &g . &k
4   ~>
5 sum j : &t , &g { &i --> j }
6             . &k { &i --> j } * &m[j]
7   where &s hasRepr MSet~Occurrence
8   where &s hasDomain `mset (..) of &t`
9   letting &m be refn(&s)
```

Figure 6.15: Vertical rule for `sum` Quantified expressions and Occurrence representation of multi-sets

this condition is posted. The third case is interesting because it does not contain a value for `&maxOccur_`, even though it is needed for the output domain. The rule assigns the value of `&maxSize_` to `&maxOccur_`, because the multi-set cannot contain any element more times than its total size. This might be a lose bound on the number of occurrences, but it is the best that can be done with the provided set of attributes.

6.2.1.1 Vertical rules

Quantified expressions over multi-set decision variables using the Occurrence representation need to be written separately for each quantifier: `forAll`, `exists`, and `sum`. This is because the elements of the representation matrix are the number of occurrences of the index value in the multi-set. For `forAll` and `exists`, the number of occurrences of a value does not

make a difference, we only need to know whether the value is present in the multi-set or not. But for `sum`, the number of occurrences of a value does matter, if a value occurs multiple times that value needs to be used multiple times in the generated quantified expression.

The rule given in Figure 6.14 is used when refining a `forAll` quantified expression over multi-sets that are represented using the Occurrence representation. The `exists` quantifier is also handled in a similar way but not presented here. It replaces a quantified expression over a multi-set decision variable into a simple quantified expression, one quantifying over an integer domain. The quantified variable `&i` represents elements in the set in the original expression, and it represents indices of the matrix in the output expression. Indices of a set with the occurrence representation correspond to elements of the set if the matrix contains a positive value in the corresponding position. A new guard is added to the output expression so the *body* of the quantified expression is only active for those values that are in the multi-set at least once.

The rule given in Figure 6.15 is for handling the `sum` case. The main difference in this rule is the guard and body of the generated quantified expression. This rule does not add a new guard, instead it uses the number of occurrences of the value as a multiplier to the body.

These vertical rules are the only rules required for representations of multi-set domains. All other multi-set operators can be refined using horizontal rules. These horizontal rules are given in Section 6.6.2.

6.2.2 Explicit representation for Multi-Sets

The Explicit representation for multi-sets uses a matrix which holds members of the multi-set. If a member occurs multiple times in the multi-set, it can be present in the matrix multiple times in different positions.

The rule given in Figure 6.16 is partial, for similar reasons to Figure 6.13. The two cases listed here are chosen to exemplify the operation of the rule.

In the first case, `size` and `maxOccur` attributes are given. The values of `&size_` is used

```
1  ~> MSet~Explicit
2  ~> matrix indexed by [int(1..&size_)] of &tau
3
4  *** mset (size &size_ , maxOccur &maxOccur_) of &tau
5  ~> forAll i : &tau .
6        (sum j : int(1..&size_) , refn[j] = i . 1) <= &maxOccur_
7
8  *** mset (size &size_ , minOccur &minOccur_) of &tau
9  ~> forAll i : &tau .
10       ((sum j : int(1..&size_) , refn[j] = i . 1) = 0) \/
11       ((sum j : int(1..&size_) , refn[j] = i . 1) >= &minOccur_)
12
13 ...
```

Figure 6.16: Explicit representation for Multi-Sets

```
1  [1000]
2
3  &quan &i in &s, &g . &k
4    ~>
5  &quan j : &r , &g { &i --> &m[j] }
6                . &k { &i --> &m[j] }
7    where &s hasType 'mset of _'
8    where &s hasRepr MSet~Explicit
9    letting &m be refn(&s)
10   letting &r be indices(&m,0)
```

Figure 6.17: Vertical rule for Quantified expressions and Explicit representation of multi-sets

in the output domain, and the values of `&maxOccur_` is used to post a structural constraint. The structural constraint ensures each value is only used as many as `&maxOccur_` in the matrix. In the second case, `maxOccur` is not given but `minOccur` is given. A similar structural constraint is posted, but this time the constraint is a disjunction of two conditions. A value is either not present in the multi-set at all, or if it is present it has to occur at least `&minOccur_` times.

6.2.2.1 Vertical rules

The rule given in Figure 6.17 is used when refining a quantified expression over a multi-set decision variables or parameters that are represented using the Explicit representation. In this rule, the quantified variable `&i` in the input expression takes values from the elements of the multi-set. However, `j` in the output quantified expression takes values from the indices of the matrix. The guard `&g` and the body `&b` of the quantified expression are written in terns of the elements of the multi-set, and the elements of the multi-set are represented using items in the Explicit matrix. For this reason the *deep-replace* construct is used to replace all references to `&i` with the corresponding expression `&m[j]`.

This vertical rule is the only rule required for representations of multi-set domains. All other multi-set operators can be refined using horizontal rules. These horizontal rules are given in Section 6.6.2.

For example the expression « `forAll i in a , i > 3 .  i in b` » will be refined to « `forAll j :  int(..)  , a'[j] > 3 .  a'[j] in b` » using this rule, where `a'` is the refinement of `a`.

6.3 Rules for function domains

6.3.1 One dimensional matrix representation

Function domains represent a mapping between values of one domain to another. The first component of a function domain is its *defined* set, and the second component is its *range* set.

This representation can be used to model function domains which are total and have an integer domain for the *defined* set. It uses a simple one dimensional matrix indexed by the *defined* set. Each item in the matrix represent a mapping in the function domain. Namely, i is mapped to the value at index i in the matrix.

Three cases of this representation selection rule are given in Figure 6.18. The first case works for a `total` function and does not post any structural constraints. The second case is

```
1  ~> Function~1D
2  ~> matrix indexed by [&fr] of &to
3    where &fr hasType 'int'
4
5  *** function (total) &fr --> &to
6
7  *** function (total, injective) &fr --> &to
8  ~> allDiff(refn)
9
10 *** function (total, surjective) &fr --> &to
11 ~> forAll i : &to . exists j : &fr . refn[j] = i
12
13 ...
```

Figure 6.18: One dimensional matrix representation

```
1  [1000]
2
3  &f(&x) ~> refn(&f)[&x]
4    where &f hasType   'function int --> _'
5    where &f hasRepr    Function~1D
```

Figure 6.19: Vertical rule for one dimensional matrix representation and the function application operator

for function domains which are `total` and `injective`. Injectivity implies distinctness, and it can be achieved by posting an `allDiff` constraint on the generated matrix. The third case is for function domains which are `total` and `surjective`. Surjectivity implies coverage of the *range* set, and it can be achieved by using a universal quantification over the values of the range set. The structural constraint ensures that there is a mapping for every value in the *range* set.

6.3.1.1 Vertical rules

The rule given in Figure 6.19 s the vertical rule for what is probably the most important operator defined for function domains: function application. Thanks to the underlying representation being a one-dimensional matrix, function application is simply translated into a matrix dereference.

```
1  [1000]
2
3  &quan &i in toSet(&f), &g . &b
4    ⇝
5  &quan k : &domFrom , &g { &i[1] --> k, &i[2] --> &m[k] }
6                      . &b { &i[1] --> k, &i[2] --> &m[k] }
7    where &f hasType    `function int --> _`
8    where &f hasDomain `function (..) &domFrom --> _`
9    where &f hasRepr     Function~1D
10   letting &m be refn(&f)
```

Figure 6.20: Vertical rule for one dimensional matrix representation and the function `toSet` operator

The rule given in Figure 6.20 s the vertical rule for quantification over the set representation of a function.

For example the expression « `forAll i in toSet(a) , i[1] > 3 .  i[2] in b` » will be refined to « `forAll j :  int(..)  , j > 3 .  a'[j] in b` » using this rule, where `a'` is the refinement of `a`.

These two vertical rules and a variation of the `toSet` rule for the `toMSet` operator are all the vertical rules needed for function domains. All other function operators can be refined using horizontal rules. These horizontal rules are given in Section 6.6.4.

6.3.2 Representing functions using relations

The representation option given in Figure 6.18 can be very efficient for some function domains. But it does not work for all function domains. This representation, given in Figure 6.21 works for all function domains by representing a function domain using a relation domain. The generated relation domain will be further refined using representation options.

Four cases of this representation selection rule are given in Figure 6.21. The first case works for function domains without any attribute. If no attribute is given for a function domain, we do not need to post many structural constraints, however by using a relation

```
1  ~> Function~AsReln
2  ~> relation of (&fr * &to)
3
4  *** function &fr --> &to
5  ~> (forAll i : &fr . (sum j in toSet(refn) . toInt(i = j[1])) <= 1)
6
7  *** function (total) &fr --> &to
8  ~> (forAll i : &fr . (sum j in toSet(refn) . toInt(i = j[1])) = 1)
9
10 *** function (total, injective) &fr --> &to
11 ~> (forAll i : &fr . (sum j in toSet(refn) . toInt(i = j[1])) = 1)
12 /\  (forAll i,j in toSet(refn) , i[1] != j[1] . i[2] != j[2])
13
14 *** function (total, surjective) &fr --> &to
15 ~> (forAll i : &fr . (sum j in toSet(refn) . toInt(i = j[1])) = 1)
16 /\  (forAll i : &to . exists j in toSet(refn) . i = j[2])
17
18 ...
```

Figure 6.21: Representing functions using relations

to model a function we still need to post a structural constraint to ensure there is at most one mapping for each value in the *defined* set. The second case works for `total` function domains. The structural constraint is very similar to that of the first case, instead of having at most one mapping for each value this time we can have exactly one mapping for each value. The third case works for `total` and `injective` function domains. In addition to the totality constraint, another structural constraint is posted to ensure injectivity. The fourth case is for `total` and `surjective` function domains. It posts an additional constraint saying there needs to be a mapping for each value in the `range` set.

6.3.2.1 Vertical rules

Refinement of function application was very straightforward for the one-dimensional matrix representation. However, when representing functions as relations it is more involved. The rule needs to be separated into pieces depending on the range type of the function. i.e. those function applications which produce a boolean expression are handled differently

```
[1000]

&f(&x) ~> forAll i in toSet(refn(&f)) , i[1] = &x . i[2]
  where &f hasType   'function _ --> bool'
  where &f hasRepr    Function~AsReln
```

Figure 6.22: Vertical rule for one dimensional matrix representation and the function application operator: `bool`

```
[1000]

&f(&x) ~> sum i in toSet(refn(&f)) , i[1] = &x . i[2]
  where &f hasType   'function _ --> int'
  where &f hasRepr    Function~AsReln
```

Figure 6.23: Vertical rule for one dimensional matrix representation and the function application operator: `int`

```
[1000]

&quan &i in &f(&x) , &guard . &body
  ~>
&quan j in toSet(refn(&f)) , &x = j[1] .
    &quan k in j[2], &guard { &i --> k }
                   . &body  { &i --> k }
  where &f hasType   'function _ --> set of _'
  where &f hasRepr    Function~AsReln
```

Figure 6.24: Vertical rule for one dimensional matrix representation and the function application operator: `set`

then those which produce a set expression.

Three examples of this kind of rule is given in Figure 6.22, Figure 6.23, and Figure 6.24. The main difference between handling booleans and integers is the quantifier being used: `forAll` and `sum` respectively.

The rule given in Figure 6.24 is only written for when the function application is in a quantified position. Thanks to horizontal rules of set domains, set expressions can always be converted to this form.

```
1  ~> Relation~IntMatrix2
2  ~> matrix indexed by [&a, &b] of bool
3    where &a hasType 'int'
4    where &b hasType 'int'
5
6  *** relation of (&a * &b)
7
8  *** relation (minSize &minSize_ , maxSize &maxSize_) of (&a * &b)
9  ~> (&minSize_ <= sum i : &a . sum j : &b . toInt(refn[i,j]))
10 /\  (&maxSize_ >= sum i : &a . sum j : &b . toInt(refn[i,j]))
11
12 *** relation (size &size_) of (&a * &b)
13 ~> &size_ = sum i : &a . sum j : &b . toInt(refn[i,j])
14
15 ...
```

Figure 6.25: Two dimensional matrix representation

6.4 Rules for relation domains

6.4.1 Two dimensional matrix representation

This representation works by converting a relation between two integer domains to a two dimensional matrix of boolean variables. It only works for relations of fixed arity and more importantly on integer domains, but it is likely to be a very good representation for many models.

Three cases of this representation selection rule are given in Figure 6.25. In the first case, the relation domain does not contain any attributes. This case does not require any structural constraints, any assignment to the matrix domain is a valid assignment to the original relation domain. In the second case, `minSize` and `maxSize` attributes are given. A conjunction of two cardinality constraints is posted to ensure this condition. The third case is similar, a `size` attribute is given and the appropriate structural constraint is posted.

```
[1000]

&r(&i,&j) ~> &refnr[&i,&j]
  where &r hasRepr RelationIntMatrix2
  where &r(&i,&j) hasType 'bool'
  letting &refnr be refn(&r)
```

Figure 6.26: Vertical rule for two dimensional matrix representation and the relation membership check operator

```
[1000]

&quan &i in toSet(&rel), &guard . &body
  ~>
&quan j1 : &index1 .
    &quan j2 : &index2
        , &guard {&i --> (j1, j2)} /\ &refnrel[j1,j2]
        . &body {&i --> (j1, j2)}

    where &rel hasRepr RelationIntMatrix2
    letting &refnrel be refn(&rel)
    letting &index1 be indices(&refnrel, 0)
    letting &index2 be indices(&refnrel, 1)
```

Figure 6.27: Vertical rule for two dimensional matrix representation and the relation `toSet` operator

6.4.1.1 Vertical rules

The rule given in Figure 6.26 is used for the membership check operator of relation domains. The input pattern is an expression which checks whether the tuple `(&i,&j)` are in the relation. The output expression is a tow-dimensional matrix dereference, using the underlying matrix which represents the relation.

The rule given in Figure 6.27 is for refining the `toSet` operator of relations. The rule is only written for when the `toSet(&rel)` expression is in a quantified position. Thanks to horizontal rules of set domains, set expressions can always be converted to this form. The output of this rule is a nested quantified expression, quantifying over all items stored in the underlying two-dimensional matrix. The original guard of the quantified expression is

```
1 ~> Relation~AsSet
2 ~> set (&attributes) of (&t_1, &t_2, ...)
3
4 *** relation (&attributes) of (&t_1, &t_2, ...)
```

Figure 6.28: Using sets to model relations

translated to work with the new quantifiers and a new guard is posted to ensure that the body is only relevant when the relation contains the quantified variables `j1` and `j2`.

6.4.2 Using sets to model relations

This representation works by converting a relation into a set of tuples. The representation selection rule needs to work for relation domains with any arity, and the rule language does not support this. As a result, the rule in implemented as a built-in rule and not in the rule language. The rule given in Figure 6.28 is for illustrative purposes only.

Internally, this rule is very simple. It only has one case as it works on any relation domain. It simply propagates all the attributes of the relation down to the set representation and converts all the components of the relation domain into components of a tuple domain that is wrapped inside the set domain.

Vertical rules of this representation are implemented internally to Conjure and not in the rule language.

6.5 Rules for partitions domains

6.5.1 Representing partitions using a multi-set of sets

This representation uses a multi-set of sets to model partitions. The outer multi-set models the separate parts of the partition, and the inner set models each part of the partition.

The implementation of this representation is separated into 3 rules, given in Figure 6.29, Figure 6.30, and Figure 6.31. The separation is required because even though each rule generates the same type, they generate different domains. When a `size` or `partSize`

```
1 ⇝ Partition~MSetOfSets
2 ⇝ mset of set of &tau
3
4 *** partition from &tau
5 ⇝ ( forAll i,j in refn, i != j . |i intersect j| = 0 )
6 /\  ( forAll i : &tau . exists s in refn . i in s )
```

Figure 6.29: Representing partitions using a multi-set of sets - no size

```
1 ⇝ Partition~MSetOfSets
2 ⇝ mset (size &n) of set of &tau
3
4 *** partition (size &n) from &tau
5 ⇝ ( forAll i,j in refn, i != j . |i intersect j| = 0 )
6 /\  ( forAll i : &tau . exists s in refn . i in s )
```

Figure 6.30: Representing partitions using a multi-set of sets – outer size known

```
1  ⇝ Partition~MSetOfSets
2  ⇝ mset (size &n) of set (size &m) of &tau
3
4  *** partition (size &n, partSize &m, ..) from &tau
5  ⇝ ( forAll i,j in refn, i != j . |i intersect j| = 0 )
6  /\  ( forAll i : &tau . exists s in refn . i in s )
7
8  *** partition (regular, size &n) from &tau
9  ⇝ ( forAll i,j in refn, i != j . |i intersect j| = 0 )
10 /\  ( forAll i : &tau . exists s in refn . i in s )
11   letting &m be domSize(&tau) / &n
12
13 *** partition (regular, partSize &m) from &tau
14 ⇝ ( forAll i,j in refn, i != j . |i intersect j| = 0 )
15 /\  ( forAll i : &tau . exists s in refn . i in s )
16   letting &n be domSize(&tau) / &m
```

Figure 6.31: Representing partitions using a multi-set of sets – both outer and inner sizes known

```
1 [1000]
2
3 &quan &i in parts(&p), &g . &k
4   ~>
5 &quan &i in refn(&p), &g . &k
6   where &p hasType 'partition from _'
7   where &p hasRepr MSetOfSets
```

Figure 6.32: Vertical rule for the `parts` operator on partitions

attribute is given for the partition, these attributes are propagated to the outer multi-set and the inner set as `size` attributes respectively.

The structural constraint in Figure 6.29 is composed of two parts. The first part of it posts the condition that parts of a partition are disjoint. The second part of it posts the condition that every value in `&tau` needs to occur in a part.

The structural constraint in Figure 6.30 is exactly the same as that of Figure 6.29. The domain pattern in the case has a `size` attribute which is used as the size attribute of the outer multi-set.

The representation selection rule in Figure 6.31 lists 3 cases. The first case has both a `size` and a `partSize` attribute. The `size` attribute is used as the `size` attribute of the outer multi-set and the `partSize` attribute is used as the `size` attribute of the inner set. The second case does not have a `partSize` attribute, but it has the `regular` attribute. The value of `partSize` is calculated using the value of the `size` attribute and the domain size of `&tau`. The third case does not have a `size` attribute, but it has the `regular` attribute. The value of `size` is calculated using the value of the `partSize` attribute and the domain size of `&tau`.

6.5.1.1 Vertical rules

The rule given in Figure 6.32 is used for the refinement of the `parts` operator. The quantification over the parts of a partition is simply turned into a quantification over the underlying multi-set which is used to represent the partition.

6.6 Horizontal rules

Horizontal rules do not change the level of abstraction of abstract decision variables. In other words, they do not make use of representation decisions. For example, a horizontal rule applied to an expression involving a decision variable with a function domain will always perform the same rewrite independent of the representation of the decision variable.

Horizontal rules are very useful in enabling Conjure's complete coverage of the Essence language. Essence contains numerous operators operating on abstract decision variables. These operators let the user of Essence write concise problem specifications.

Most operators in Essence can be defined in terms of other operators in the language. A simple example to this is the super-set ($\supset$) operator for sets, which is an arguments flipped version of the subset operator ($\subset$) for sets. A slightly more involved example is set equality ($=$), which can be viewed as a conjunction of two operators, subset-or-equal ($\subseteq$) and superset-or-equal ($\supseteq$) for set.

Horizontal rules provide a correct way to refine many operators. They reduce the number of operators and language constructs that are required to be implemented via vertical rules. Reducing the need for vertical rules is important because vertical rules are defined depending on specific representations and adding a new representation requires the addition of vertical rules.

In the rest of this section some example horizontal rules are given.

6.6.1 Horizontal rules for set domains

This subsection gives some of the most important horizontal rules for set domains.

6.6.1.1 Set cardinality

Figure 6.33 gives the generic rule for set cardinality. It makes use of a `sum` quantified expression over the set variable.

```
[1000]

|&s| ~> sum i in &s . 1
  where &s hasType 'set of _'
```

Figure 6.33: Set cardinality

```
[500]

|&s| ~> &size_
  where &s hasDomain 'set (size &size_ ,..) of _'
```

Figure 6.34: Set cardinality for fixed size sets

```
[1000]

&a = &b ~> &a subsetEq &b /\ &a supsetEq &b
  where &a hasType 'set of _'
  where &b hasType 'set of _'
```

Figure 6.35: Set equality to subsets

6.6.1.2 Set cardinality for fixed size sets

Figure 6.34 gives a better rule for sets with a size attribute attached to their domains. It uses the `hasDomain` predicate of the rule language to extract the value of the size attribute. The replacement expression does not contain a reference to the original decision variable any more. This rule is especially useful, because expressions involving the cardinality operator will be generated by other rules and they can be *compiled away* for fixed size sets.

6.6.1.3 Set equality to subsets

Figure 6.35 gives a rule which replaces a set equality expression with a conjunction of two expressions one using `subsetEq` and one using `supsetEq`. This is almost a textbook definition of set equality, it is trivially correct and can be used by all set representations.

```
[900]

&a = &b
  ~>
(forAll i : &t1 . i in &a <-> i in &b) /\
(forAll i : &t2 . i in &a <-> i in &b)
  where &a hasDomain `set (..) of &t1`
  where &b hasDomain `set (..) of &t2`
  where &t1 hasType `int`
  where &t2 hasType `int`
```

Figure 6.36: Set equality: an alternative

```
[1000]

&e in &s ~> exists j in &s . j = &e
  where &s hasType `set of _`
```

Figure 6.37: Set membership

6.6.1.4 Set equality: an alternative

Figure 6.36 gives an alternative rule to handle set equality. This rule is potentially much better than the rule in Figure 6.35 however it is only applicable if both operands are sets of integers. The rule in Figure 6.35 is more widely applicable since it does not have any guards on the actual domain of the operands. Since this horizontal rule given better models in general it is placed at a lower level than the other one. This stops Conjure from applying both rules and generating alternative models, instead Conjure tries to apply the more specific rule when it can and only uses the more generic rule otherwise. For experimentation purposes the level of this rule can be changed to `1000`, which will force Conjure to create models using both rules.

6.6.1.5 Set membership

Set membership can be represented using an `exists` quantifier over the set. The rule given in Figure 6.37 implements this transformation.

```
[1000]

&a subset &b ~> &a subsetEq &b /\ &a != &b
  where &a hasType 'set of _'
  where &b hasType 'set of _'
```

Figure 6.38: Strict subset in terms or subset-or-equal and inequality of sets

```
[1000]

&a subset &b ~> &a subsetEq &b /\ |&a| < |&b|
  where &a hasType 'set of _'
  where &b hasType 'set of _'
```

Figure 6.39: Strict subset in terms or subset-or-equal and cardinality comparison

```
[1000]

&a subsetEq &b ~> forAll i in &a . i in &b
  where &a hasType 'set of _'
  where &b hasType 'set of _'
```

Figure 6.40: Subset-or-equal in terms of quantified expressions

6.6.1.6 Subset and related operators

Figure 6.38 gives a rule which implements the strict `subset` operator in terms of the `subsetEq` operator. This rule produces a conjunction of two constraints and posts the additional condition that the sets are distinct. Figure 6.39 gives an alternative rule to implement the same `subset` operator. In this rule, instead of introducing a complicated set inequality constraint, cardinalities of the two sets are constrained. This can be particularly powerful if one or both of the sets have a fixed cardinality. Because if that is the case, the rule in Figure 6.34 will lookup the statically known cardinalities of the sets and produce a much smaller constraint than its alternative. Figure 6.40 gives a rule which turns a `subsetEq` operator into a quantified expression. Other subset related operators, like `supset` and `supsetEq` are implemented similarly.

```
1 [1000]
2
3 exists &i in &a union &b, &guard . &body
4   ~>
5 (exists &i in &a, &guard . &body) \/
6 (exists &i in &b, &guard . &body)
7   where &a hasType 'set of _'
8   where &b hasType 'set of _'
```

Figure 6.41: `exists` quantification over sets

```
1 [1000]
2
3 max(&a union &b) ~> max(max(&a),max(&b))
4   where &a hasType 'set of _'
5   where &b hasType 'set of _'
```

Figure 6.42: Maximum of the union of two sets

6.6.1.7 `exists` quantification over sets

Figure 6.41 gives a rule which handles the case when the input expression in an existential quantification over the union of two sets. It separates the quantified expression into two quantified expression one for each set operand of the union; and the two expressions are combined using a disjunction. This way, the rule avoids introducing an auxiliary decision variable with a set domain.

6.6.1.8 Set `max` operator

Figure 6.42 gives a rule which handles the case when the `max` operator is applied to the union of two set expressions. It implements this operator by using the `max` operator defined on integers, which takes two integers and evaluates to the maximum value out of these two. On the other hand, Figure 6.43 gives a rule which works by introducing an auxiliary decision variable. The decision variable `aux` has the same domain as the elements of the set, and it is constrained to be greater than or equal to each element in the set.

```
[1000]

max(&s)  ~> { aux
            @ find aux : &tau
              such that
                forAll i in &s . i <= aux,
                aux in &s
            }
  where &s hasDomain `set (..) of &tau`
```

Figure 6.43: Maximum value in an atomic set

```
[1000]

|&s| ~> sum i in &s . 1
  where &s hasType `mset of _`
```

Figure 6.44: Cardinality of multi-sets

6.6.2 Horizontal rules for multi-set domains

This subsection gives some of the most important horizontal rules for set domains.

6.6.2.1 Cardinality of multi-sets

Figure 6.44 gives the horizontal rule which refines the cardinality operator. The rule is very similar to one of the set cardinality rules, however the type pattern in the guard is different in this rule.

Several rules will be omitted here because of their similarity to the set rules. For example: rules for handling `subset`, `supset`, `supsetEq`, multi-set equality are omitted.

6.6.3 Multi-set frequency operator

Figure 6.45 gives the rule which implements the `freq` operator on multi-sets in terms of a `sum` quantified expression. `freq(m,x)` returns the number of occurrences of the value `x` in the multi-set `m`, and this refinement rule directly follows the definition of the operator.

```
[1000]

freq(&m,&x) ~> sum i in &m . toInt(i = &x)
  where &m hasType `mset of _`
```

Figure 6.45: Frequency operator `freq` of multi-sets

```
[1000]

&a subsetEq &b ~> ( forAll i in &a . freq(&a,i) <= freq(&b,i) )
                  /\ ( forAll i in &b . freq(&a,i) <= freq(&b,i) )
  where &a hasType `mset of _`
  where &b hasType `mset of _`
```

Figure 6.46: Subset-or-equal for multi-set domains

```
[1000]

|&f| ~> |toSet(&f)|
  where &f hasType `function _ --> _`
```

Figure 6.47: Cardinality of a function

6.6.3.1 Subset-or-equal for multi-set domains

Figure 6.46 gives the rule which implements `subsetEq` for multi-set domains. This refinement rule is considerably different from the corresponding refinement rule for set domains because it needs to make sure not only that every value in the first multi-set is in the second multi-set, but also that the number of occurrences of those values line up properly. The rule makes use of a `forAll` quantified expression and the `freq` operator to this effect.

6.6.4 Horizontal rules for function domains

This subsection gives some of the most important horizontal rules for function domains.

```
1 [800]
2
3 &f(&x) = &y ~> forAll i in toSet(&f) , i[1] = &x . i[2] = &y
4   where &f hasType 'function _ --> _'
```

Figure 6.48: Function application in an equality context

6.6.4.1 Cardinality of a function

Figure 6.47 gives the rule which implements cardinality of function expressions. It is very simple to get a correct implementation of cardinality of functions, because functions have a `toSet` operator defined on them. The cardinality of a function is equal to the cardinality of the set representation of the same function.

6.6.4.2 Function application in an equality context

This rule, Figure 6.48 is only applicable when the original expression is in the form of an equality and has as one of its operands a function application. It works by producing a quantified expression over the function variable, more accurately over the set representation of the function variable. In the body of the quantified expression `i` is a tuple which represents a mapping. The second component of this tuple is constrained to be equal to `&y`, only when the first component is equal to `&x`.

6.6.4.3 Quantify over `defined` values in a function

Figure 6.49 gives a very important horizontal rule. Writing quantified expressions over all values that are defined for a function variable is very common. Thankfully, it can be implemented using a quantification over the set representation of a function variable. This way each function representation only needs to give a vertical rule for the `toSet` operator, and the `defined` operator works without any more work.

```
[1000]

&quan &i in defined(&f), &g . &b

    ⇝

&quan j in toSet(&f) , &g { &i --> j[1] }
                     . &b { &i --> j[1] }

    where &f hasType    `function _ --> _`
```

Figure 6.49: Quantify over `defined` values in a function

```
[1000]

&f = &g ⇝ forAll i in defined(&f) . &f(i) = &g(i)
  where &f hasType `function _ --> _`
  where &g hasType `function _ --> _`
```

Figure 6.50: Equality of functions

```
[1000]

inverse(&f, &g)
  ⇝
(forAll i in toSet(&f) . (i[2],i[1]) in toSet(&g))
  /\
(forAll i in toSet(&g) . (i[2],i[1]) in toSet(&f))
  where &f hasType `function _ --> _`
  where &g hasType `function _ --> _`
```

Figure 6.51: Function inverse

6.6.4.4 Equality of functions

Figure 6.50 gives a rule which makes use of quantification over `defined` values of a function. In order for `&f` and `&g` to be equal, for all values `&f` is defined for, `&g` needs to be map the same value to the same result.

```
1 [1000]
2
3 &a = &b ⇝ toSet(&a) = toSet(&b)
4   where &a hasType 'relation'
5   where &b hasType 'relation'
```

Figure 6.52: Relation equality

```
1 [1000]
2
3 &a in &b ⇝ &a in toSet(&b)
4   where &b hasType 'relation'
```

Figure 6.53: Relation membership

6.6.4.5 Function inverse

Figure 6.51 gives a rule which refines the `inverse` operator on functions. This operator takes two functions as arguments and evaluates to a boolean, indicating whether the two functions are inverses of one another or not. The implementation of this rule uses quantification over the set representation of both functions and checks for membership in the body of the quantified expressions.

6.6.5 Horizontal rules for relation domains

This subsection gives some of the most important horizontal rules for relation domains.

Many operators on relation domains treat relations as set-like containers. Hence, most of the horizontal rules implement their required functionality using existing set operators.

6.6.5.1 Relation equality

Figure 6.52 gives a rule for relation equality. This rule is implemented by checking for equality between the set representations of both relations.

```
1 [1000]
2
3 &a = &b ~> ( forAll i in parts(&a) . i in parts(&b) ) /\
4            ( forAll i in parts(&b) . i in parts(&a) )
5   where &a hasType `partition from _`
6   where &b hasType `partition from _`
```

Figure 6.54: Partition equality

6.6.5.2 Relation membership

In a similar manner to other horizontal rules for relation domains, the rule given by Figure 6.53 is implemented in terms of the set representation of the relation.

6.6.6 Horizontal rules for partition domains

Partition domains generally act like nested set domains. The operators defined on partitions are mostly set operators and those are handled using the horizontal rules for sets.

This subsection gives one example horizontal rules for partition domains.

This rule, Figure 6.54, implements equality checking between two partition expressions. The implementation of the rule uses a quantification over the `parts` of a partition. The `parts` operator returns a set of parts of the partition, moreover each part is also a set of item contained in the partition. In the body of the quantified expressions, set membership operator is used to check whether a part of one partition is also a part of another partition. Partitions are considered equal if they contain the same parts.

6.6.7 Horizontal rules for decomposition

Horizontal rules can be used to give decompositions of global constraints, too. As an example, Figure 6.55 gives the decomposition for the `allDiff` constraint. The rule is very simple, it merely gives the decomposition of `allDiff` into a clique of inequality constraints using the familiar Essence syntax with meta-variables. It also uses guards to control when this rule is applicable: it is generally not desired to decompose `allDiff` when the argument

```
1 allDiff(&m) ~> forAll i,j : &r, i < j . &m[i] != &m[j]
2   where !(&m hasType    'matrix indexed by [_] of int')
3   where   &m hasDomain 'matrix indexed by [&r] of _'
```

Figure 6.55: Decomposition of `allDiff`

is a matrix of integers. However, since the output language Essence′ does not contain `allDiff` for any other type other than integers, `allDiff` on all other types need to be decomposed into a clique of inequalities. If the matrix contains abstract domains in it, the generated inequality constraints will be further refined using other rules.

6.7 Summary

This chapter gave a listing of rules of Conjure. The first part of the chapter contains a section for each abstract type of Essence, and each section contains a subsection for each representation option. The second part of the chapter contains a listing of an interesting subset of of the horizontal rules. Horizontal rules are representation independent and they do a lot of hard work: Thanks to having horizontal rules, each representation needs to provide a very small number of vertical rules.

Chapter 7

Extensibility

This chapter demonstrates the extensibility of Conjure by giving two complete examples of adding new representations: one for set domains and one for function domains.

7.1 Adding the Gent representation

The Gent representation [Jef+05] was designed to try to combine the strengths of the occurrence and explicit representations. It works for set of integers using a one-dimensional matrix. Items in the matrix take values from 0 to maximum possible cardinality of the set. The membership of an index value of the matrix is denoted by a non-zero item in the corresponding position. i.e., i is considered to be in the set if and only if $M[i] > 0$, if M is the Gent representation of a set variable.

Moreover, the Gent representation requires an additional condition. The non-zero value denoting membership cannot be just a free non-zero integer, the non-zero items in the matrix has to be in a strictly increasing order.

The rule given in Figure 7.1 is the representation selection rule for Gent representation. It contains 4 cases, and omits the cases where the set cardinality is known. In the first two of these cases, the `maxSize` value is immediately available as a part of the domain, in the last two cases it can be calculated using the `domSize` operator on the inner domain `&tau`. In the first and the third case, the `minSize` attribute is given as a part of the domain. In these

```
1  ~> Set~Gent
2  ~> matrix indexed by [&tau] of int(0..&maxSize_)
3  ~> forAll i : &tau .
4          (refn[i] = 0) \/
5          (refn[i] = 1 + sum j : &tau, j < i /\ refn[j] > 0 . 1)
6
7    where &tau hasType 'int'
8
9  *** set (minSize &minSize_ , maxSize &maxSize_) of &tau
10     ~> &minSize_ <= sum i : &tau . toInt(refn[i] != 0)
11
12 *** set (maxSize &maxSize_) of &tau
13
14 *** set (minSize &minSize_) of &tau
15     ~> &minSize_ <= sum i : &tau . toInt(refn[i] != 0)
16     letting &maxSize_ be domSize(&tau)
17
18 *** set of &tau
19     letting &maxSize_ be domSize(&tau)
```

Figure 7.1: Representation selection rule for Gent representation of sets.

cases, additional structural constraints are posted to require this condition.

The output domain from this rule is, as required by the definition of the representation, a one-dimensional matrix domain indexed by `&tau`. `&tau` is the meta-variable for matching the inner part of a set domain. The elements of the matrix have an integer domain from o up to maximum cardinality. The structural constraint posts the condition that for each index `i` in the matrix, the value of `refn[i]` has to be either `0` indicating that `i` is not a member of the set; or it has to be equal to the number of non-zero items in the matrix up to this point plus one. This structural constraint ensures the main invariant of the Gent representation, that the non-zero values in the matrix has to be in strictly increasing order.

The rule given in Figure 7.2 is used when refining quantified expressions over sets represented using the Gent representation. In this rule the quantified variable `&i` in the input expression takes values from the elements of the set. In the output `&i` takes values from the indices of the Gent matrix. However, this does not present a problem because the two domains are the same. The original guard part of the input expression is kept

```
1 [1000]
2
3 &quan &i in &s , &guard . &body
4     ~>
5 &quan &i : &t , &guard /\ &m[&i] > 0 . &body
6   where &s hasDomain `set (..) of &t`
7   where &s hasRepr Set~Gent
8   letting &m be refn(&s)
```

Figure 7.2: Vertical rule for Quantified expressions and the Gent representation of sets

```
1 [900]
2
3 &x in &s ~> refn(&s)[&x] > 0
4   where &s hasRepr Set~Gent
```

Figure 7.3: Vertical rule for better membership check in the Gent representation

unchanged and a new guard is posted. This new guard makes sure that the body part of the quantified expression is only used then `&i` is indeed found in the set.

For example the expression « `forAll i in a, i > 3 . i in b` » will be refined to « `forAll i : int(..) , i > 3 /\a'[i] > 0 . i in b` » using this rule, where `a'` is the refinement of `a`.

This is sufficient to refine all set operators, thanks to horizontal rules for set domains (Section 6.6.1). However, some operators can be refined to better constraints once they are specialised to the Gent representation. One example of such an operator is `in`, the membership predicate for sets.

The rule given in Figure 7.3 is a vertical rule for better handling the set membership operator, `in`. It simply checks the name of the representation for `&s` and produces the output expression using a matrix dereference and checking for whether the value if positive or not. Without this rule Conjure would still produce a correct refinement of the `in` operator. However then, it would have to generate a more verbose expression and several unnecessary constraints.

For example the expression « `x in a` » will be refined to « `a'[x] > 0` » using this rule,

```
language Essence 1.3
find a,b : set (minSize 2, maxSize 4) of int(0..9)
find c   : set (minSize 3, maxSize 4) of int(0..9)
such that c subsetEq a union b, 1 in a
```

Figure 7.4: Example problem using set variables.

```
language ESSENCE' 1.0

find a_SetGent, b_SetGent, c_SetGent:
        matrix indexed by [int(0..9)] of int(0..4)
such that
  forAll i : int(0..9) .
      c_SetGent[i] > 0
      ->
      (exists j : int(0..9) . a_SetGent[j] > 0 /\ j = i)
      \/
      (exists j : int(0..9) . b_SetGent[j] > 0 /\ j = i) ,
  a_SetGent[1] > 0,
  ...
```

Figure 7.5: Example problem using set variables, refined using Gent representation.

where `a'` is the refinement of `a`. Without this rule it would have been refined to

`« exists i : int(..) . a'[i] > 0 . x = 4 »`.

7.1.1 Example

This section gives a simple problem specification which has three set variables and only one constraint. The Essence specification is given in Figure 7.4. The final model using the Gent representation for all three set decision variables if given in Figure 7.5. The structural constraints are left out from this model to focus on the refinement of the original constraint.

It takes 6 rule applications for the constraint to be fully refined. In what follows, the inputs and outputs of each step is given in successive figures, see Figures 7.6 to 7.12. These figures are takes directly out of Conjure and the identifier names starting with "`v__`" are auto-generated unique names as described in Section 5.7.

```
c subsetEq a union b
  ⇝
(forAll v__10 in c . v__10 in a union b)
```

Figure 7.6: Step 1: Applying rule Figure 6.40

```
v__10 in a union b
  ⇝
(exists v__11 in a union b . v__11 = v__10)
```

Figure 7.7: Step 2: Applying rule Figure 6.37

```
(exists v__11 in a union b . v__11 = v__10)
  ⇝
(exists v__11 in a . v__11 = v__10) \/
(exists v__11 in b . v__11 = v__10)
```

Figure 7.8: Step 3: Applying rule Figure 6.41

7.2 A new representation for partial function domains

This section presents a new representation for partial function domains: `1DPartial`. Function domains are used to model mappings between two sets: the `defined` and the `range` set. The `1DPartial` representation is specialised to partial function domains where the `defined` set is of type integer. It uses a matrix which contains an entry for every possible mapping in the function domain together with a boolean marker denoting whether the entry should be regarded as a part of the final value of the function or not.

The rule given in Figure 7.13 is part of the representation selection rule for `1DPartial`. This rule doesn't contain cases for when the function domain has a `total` attribute; because for total function domains, the representation using a one-dimensional array without boolean

```
(exists v__11 in a . v__11 = v__10)
  ⇝
(exists v__12 : int(0..9) , a_Set~Gent[v__12] > 0 . v__12 = v__10)
```

Figure 7.9: Step 4: Applying rule Figure 7.2

```
(exists v__11 in b . v__11 = v__10)
  ⇝
(exists v__13 : int(0..9) , b_Set~Gent[v__13] > 0 . v__13 = v__10)
```

Figure 7.10: Step 5: Applying rule Figure 7.2

```
(forAll v__10 in c .
  (exists v__12 : int(0..9) , a_Set~Gent[v__12] > 0
                            . v__12 = v__10) \/
  (exists v__13 : int(0..9) , b_Set~Gent[v__13] > 0
                            . v__13 = v__10))
  ⇝
(forAll v__14 : int(0..9) , c_Set~Gent[v__14] > 0 .
  (exists v__12 : int(0..9) , a_Set~Gent[v__12] > 0
                            . v__12 = v__14) \/
  (exists v__13 : int(0..9) , b_Set~Gent[v__13] > 0
                            . v__13 = v__14))
```

Figure 7.11: Step 6: Applying rule Figure 7.2 once more

```
1 in a ⇝ a_Set~Gent[1] > 0
```

Figure 7.12: Step 7: Applying rule Figure 7.3 once more

markers – as given in Figure 6.18 – is superior.

This representation requires the addition of 2 vertical rules. The first vertical rule is given in Figure 7.14, and it handles the function application operator. It uses bubbles as described in Section 5.6 to post a constraint on the boolean marker for when the value in the second component of the tuple is referenced. Then automatic handling of bubbles by Conjure makes sure that the function is defined when needed. The second vertical rule is given in Figure 7.15 and handles quantification over all the mappings in an with a function domain. The guard (`&g`) and the body (`&b`) of the input expression are modified to use the new quantifier variable `k`. Moreover, an additional guard is posted so those values which contain a `false` marker are filtered out.

```
1  ~> Function~1DPartial
2  ~> matrix indexed by [&fr] of (bool,&to)
3    where &fr hasType 'int'
4
5  *** function &fr --> &to
6
7  *** function (injective) &fr --> &to
8  ~> forAll i,j : &fr , i != j /\ refn[i,1] /\ refn[j,1]
9                  . refn[i,2] != refn[j,2]
10
11 *** function (surjective) &fr --> &to
12 ~> forAll i : &to . exists j : &fr , refn[j,1] . refn[j,2] = i
13
14 ...
```

Figure 7.13: Representation selection rule for `1DPartial` representation of functions.

```
1  &f(&x) ~> { refn(&f)[&x,2] @ such that refn(&f)[&x,1] }
2    where &f hasType   'function int --> _'
3    where &f hasRepr    Function~1DPartial
```

Figure 7.14: Vertical rule for function application and the `1DPartial` representation

```
1  &quan &i in toSet(&f), &g . &b
2    ~>
3  &quan k : &domFrom
4       , &g { &i[1] --> k, &i[2] --> &m[k,2] } /\ &m[k,1]
5       . &b { &i[1] --> k, &i[2] --> &m[k,2] }
6    where &f hasType   'function int --> _'
7    where &f hasDomain 'function (..) &domFrom --> _'
8    where &f hasRepr    Function~1DPartial
9    letting &m be refn(&f)
```

Figure 7.15: Vertical rule for the `toSet` operator and the `1DPartial` representation

```
1  language Essence 1.3
2
3  given n : int
4  letting ROW, COL be domain int(1..n)
5
6  find queenAtRow : function (injective) ROW --> COL
7
8  minimising |queenAtRow|
9
10 such that
11  forAll (r1,c1),(r2,c2) in toSet(queenAtRow)
12   , r1 < r2
13   . |c1-c2| != |r1-r2|,
14
15  forAll r : ROW
16   , !(r in defined(queenAtRow))
17   . forAll c : COL .
18      (exists r2 : ROW , r != r2 . queenAtRow(r2) = c) \/
19      (exists r2 : ROW , r != r2 . |queenAtRow(r2) - c| = |r2 - r|)
```

Figure 7.16: The Essence specification of the Dominating Queens problem

7.2.1 The Dominating Queens Problem

An Essence problem specification for the Dominating Queens Problem[Gib+97] is given in Figure 7.16. This problem specification contains a single decision variable which has a partial function domain. Hence its refinement can use the newly added `1DPartial` representation for partial function domains.

The problem is placing the minimum number of queens of a chess board such that no two queens attack each other and at least one queen attacks every empty cell.

In the problem specification, the decision variable `queenAtRow` possibly contains an entry for every row. If it does contain an entry, the image of the function is the column at which a queen is present, if it doesn't the row doesn't contain any queens. The `injective` attribute on the domain of `queenAtRow` posts the constraint that queens need to be placed on separate columns. The problem has 2 constraints. The first constraint makes sure that the queens are not on the same diagonal. The second constraint makes sure for all rows without a queen,

```
language ESSENCE' 1.0

given n: int
find queenAtRow_active: matrix indexed by [int(1..n)] of bool
find queenAtRow_value: matrix indexed by [int(1..n)] of int(1..n)
minimising sum i : int(1..n) . toInt(queenAtRow_active[i])
such that
  forAll i,j : int(1..n) .
    (i != j /\ queenAtRow_active[i] /\ queenAtRow_active[j])
    -> (queenAtRow_value[i] != queenAtRow_value[j]),
  ...
```

Figure 7.17: The Essence′ model for the Dominating Queens problem using the `1DPartial` representation

and for all cells on such a row there must be a queen at the same column or at a diagonal attacking this empty cell. Conjure generates the Essence′ model partly given in Figure 7.17 using the `1DPartial` representation.

7.3 Summary

This chapter demonstrates how easy it is to add two new representation from scratch. In order to have a fully working variable representation in the first representation, we only had to provide one representation decision rule and one vertical rule for handling quantified expressions. In addition, we have also seen how to provide a specialised rule for the `in` operator and placed it at a lower precedence level than the usual `1000`. This means the rule given in Figure 7.3 will be applied before applying the horizontal rule Figure 6.37. Comparing Figure 7.7 and Figure 7.12 is a good example showing this precedence mechanism in action: the former uses the generic rule because the right-hand side is not an atomic set variable with the Gent representation and the latter uses the more specific rule because `a` is an atomic set variable represented using the Gent representation. The second newly added representation is one for function domains, and in particular partial function domains. This representation improves on Figure 6.21 in the particular case when the `defined` set is of

type integer. Two vertical rules are also presented for this representation in order to have a fully working variable representation.

Chapter 8

Symmetry Breaking

This chapter explains an automated symmetry breaking technique implemented in Conjure. In short, this technique works by introducing two operators $\dot{\leq}$ and $\dot{<}$ for ordering abstract decision variables of Essence. Structural constraints of representation selection rules are then modified to make use of these operators to eliminate symmetry as soon as it enters the model. The technique applies to arbitrarily nested symmetries and represents a significant step forward for automated constraint modelling.

Many constraint problems contain symmetry, which can lead to redundant search [Gen+99; Fle+02b]. If a partial assignment is shown to be invalid, the solver will be wasting time if it ever considers a symmetric equivalent of it. Much symmetry enters constraint models through the process of constraint modelling [Fri+03]. Conjure exploits this by breaking symmetry as it enters the model. This obviates the need for an expensive symmetry detection step following model formulation, as used by other approaches [Man+05; Mea+11]. The added symmetry breaking constraints hold for the entire parameterised problem class — not just a single problem instance.

8.1 Breaking symmetry as soon as it enters the model

Symmetry enters constraint models in two ways. Some problems have inherent symmetries, which if not broken get reflected in the model. Other symmetries are introduced by the

```
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched: set (size w) of
                partition (regular, size g) from Golfers

such that
  forAll week1, week2 in ached, week1 != week2 .
    forAll group1 in parts(week1) .
      forAll group2 in parts(week2) .
        |group1 intersect group2| < 2
```

Figure 8.1: Essence specification of the Social Golfers Problem

modelling process; in this case a single solution to the problem corresponds to multiple assignments to the variables of the model. We call these *model* symmetries. As an example, consider the Social Golfers Problem (Figure 8.1), which requires finding a set of w partitions. If this set is modelled as an array indexed by $1..w$ then all $w!$ permutations of the array correspond to the same set. This symmetry is introduced when an arbitrary decision is made about which set element goes in which cell of the array. Similarly, if the $g * s$ *Golfers* are modelled by the integers $1..g * s$ then $g * s$ symmetries are introduced because of the arbitrary decision of which golfer corresponds to which integer. The problem-specification language Essence has been designed so that, unlike other modelling languages, problems can be specified without having to make the arbitrary decisions that introduce model symmetries.

Our view is that a modeller, human or machine, should be aware of the modelling decisions it makes and thus know what symmetries it introduced into the model. Indeed, this should be the case whether modelling an entire problem class or a single problem instance. Hence there is no need to apply sophisticated methods to the generated model to detect symmetries introduced by the modelling process.

Frisch et al. [Fri+07] show how each modelling rule of Conjure can be extended to generate a description of the symmetries it introduces and how the generated descriptions can be composed to form a description of the symmetries introduced into the model. The

```
1 ~> Set~Explicit~Sym
2 ~> matrix indexed by [int(1..&n)] of &tau
3
4 *** set (size &n, ..) of &tau
5 ~> allDiff(refn)
```

Figure 8.2: Representation selection rule without Symmetry breaking

intention was that the resulting description could then be used to generate symmetry-breaking constraints to add to the model, though these descriptions were never fully developed into a method for automatically generating symmetry-breaking constraints.

The current version of Conjure takes a different approach to generating symmetry-breaking constraints: each rule that introduces symmetries also generates a constraint to break those symmetries. There is only one rule in Conjure which does not break all symmetry which it introduces – the rule that refines an unnamed type, such as *Golfers*, to a range of integers. For unnamed types we do not yet have a general symmetry-breaking method. We provide total symmetry breaking for all other model symmetries.

To illustrate how Conjure rules can be extended to generate symmetry-breaking constraints, consider the rule to build the explicit representation of a set given in Figure 8.2.

This rule transforms a set of a size n into a matrix of with n index values, where each value in the matrix is a member of the set. A constraint is imposed to ensure that the cells of the matrix are all different. For any `tau` other than integers or booleans, we have to further decompose the `allDiff` constraint into $O(n^2)$ not-equal constraints.

Now consider extending this rule to generate a constraint to break the symmetry it introduces, that the index values of the matrix can be permuted in any way. The simplest way to break this symmetry is to impose a total order on the elements of the matrix. As the elements of the matrix can be any type `tau` we introduce two new operators, $\dot{\leq}$ and $\dot{<}$. These operators provide a total ordering (and a strict version of the same total ordering) for all types in Conjure. These orderings are not intended to be "natural" and are not available to Essence users. They are used only in refinement rules to generate effective symmetry-

```
1 ~> Set~Explicit
2 ~> matrix indexed by [int(1..&n)] of &tau
3
4 *** set (size &n, ..) of &tau
5 ~> forAll i : int(1..&n-1) . refn[i] .< refn[i+1]
```

Figure 8.3: Representation selection rule with Symmetry breaking

breaking constraints. Using these orderings, the `Set~Explicit~Sym` rule is modified to a rule that breaks all the symmetries it introduces. (Figure 8.3)

Rather than introducing a chain of $\dot{\leq}$ constraints, this rule exploits the fact that the elements of the set are required to be all different and strengthens the ordering to $\dot{<}$ constraint. This replaces $O(n^2)$ not-equal constraints with only $O(n)$ $\dot{<}$ constraints.

Other refinement rules can exploit the fact that symmetry breaking is performed immediately to produce more efficient refinements. Consider refining the constraint $S = T$, representing S and T as matrices S' and T' with the `Set~Explicit~Sym` representation. To find if S' and T' represent the same set we must check if each element of S' is equal to *any* element of T', since the order of elements in the matrices can be different. However, when the `Set~Explicit` representation is used we know that $S = T$ if and only if $S' = T'$, because each assignment of S is represented by exactly one assignment to S' that satisfies the symmetry breaking constraint. This gives a much smaller constraint, which propagates much more effectively.

We illustrate the new approach to symmetry-breaking by showing how the SGP specification (Figure 8.1) is refined into a model with symmetry-breaking constraints. We consider generating only one model. To focus on the issues of concern, we consider only how the decision variables are refined, ignoring all constraints other than symmetry-breaking constraints. First, Conjure replaces `type of size g*s` with `int(1..g*s)`:

```
1 given w, g, s : int(1..)
2 find sched' : set (size w) of
3                 partition (regular, size g) from int(1..g*s)
```

After this, Conjure refines the type of the decision variable starting by rewriting the outer `set` constructor using the `Set~Explicit` rule given in the previous section. This generates the following refinement.

```
given w, g, s : int(1..)
find sched' : matrix indexed by [int(1..w)] of
                partition (regular, size g) from int(1..g*s)
such that
  forAll i : int(1..w-1) .
    sched'[i] .< sched'[i+1]
```

This refinement step shows all of the important features of our method. We have introduced a new, compact constraint which both breaks symmetry, and ensures all members of the matrix are distinct. We next transform the partition into a set of sets:

```
given w, g, s : int(1..)
find sched'' : matrix indexed by [int(1..w)] of
                set (size g) of set (size (g*s)/g) of int(1..g*s)
such that
  forAll i : int(1..w-1) .
    sched''[i] .< sched''[i+1],
  forAll j : int(1..w-1) .
    forAll k1,k2 : sched''[j] , k1 != k2 .
      | k1 intersect k2 | = 0
```

This refinement does not appear to have changed the symmetry breaking constraint but it has in fact been refined from a partition to a set of sets. We have also added a constraint to impose that each cell of the partition is distinct. As we are just considering symmetry breaking, we will not consider further this structural constraint, which constrains the sets to be disjoint. We now apply `Set~Explicit` again.

```
1 given w, g, s : int(1..)
2 find sched''' : matrix indexed by [int(1..w), int(1..g)] of
3                 set (size (g*s)/g) of int(1..g*s)
4 such that
5   forAll i : int(1..w-1) .
6     sched'''[i,..] .< sched'''[i+1,..],
7   forAll j : int(1..w) .
8     forAll k : int(1..g-1) .
9       sched'''[j,k] .< sched'''[j,k+1]
```

The first constraint here is the refined version of the already existing symmetry breaking constraint. Once again by design the $\lessdot$ constraint maps naturally to the matrices used in refinement. The second constraint is the symmetry breaking on matrix of sets, now transformed into a matrix of matrices. We use the same refinement rule, even though we are now refining a set inside a matrix. Conjure automatically deals with the array indices, and inserts the outer `forAll j : int(1..w)`, in a process called *lifting*. To finish we apply `Set~Explicit` once more, and also change all the $\dot{<}$ and $\dot{\leq}$ constraints into their final form – lexicographic ordering constraints on matrices and ordering on integers.

If we had not broken symmetry immediately, but used the `Set~Explicit~Sym` representation, the constraints which impose that each partition in the outermost set is different would now be very complex, rather than the simple and efficient ordering constraints which we have generated. This shows the benefit of breaking symmetries as soon as they are introduced, rather than delaying and using a general technique for symmetry breaking after model generation is finished.

```
given w, g, s : int(1..)
find sched''' : matrix indexed by [int(1..w),
                    int(1..g),int(1..(g*s/g))] of int(1..g*s)
such that
  forAll i : int(1..w-1) .
    sched'''[i,..,..] <lex sched'''[i+1,..,..],
  forAll j : int(1..w) .
    forAll k : int(1..(g*s)/g-1).
      sched'''[j,k,..] <lex sched'''[j,k+1,..],
  forAll j : int(1..w) . forAll k in int(1..g)
    forAll l : int(1..(g*s)/g-1) .
      sched'''[j,k,l] < sched'''[j,k,l+1]
```

8.2 Implementation of the ordering operators

The two operators introduced in this chapter are $\dot{\leq}$ and $\dot{<}$. Conjure needs to contain expression refinement rules to implement these operators for the built-in types of Essence —booleans, integers, enumerated types, tuples and matrices— in addition to the abstract type constructors. The implementation of these operators depend on the representation of an abstract type constructor, hence each representation needs to provide vertical rules in order to handle them.

Booleans, integers and enumerated types are ordered types. Hence the standard ordering operators $<$ and $<=$ are defined for them. The two symmetry ordering operators, $\dot{<}$ and $\dot{\leq}$ are simply implemented using $<$ and $<=$ respectively.

For tuples and matrices, a decomposition of lexicographical ordering is used. That is, for two tuples a and b of any arity, either the first component of both tuples are ordered or they are equal and the rest of the tuple is lexicographically ordered. The rest of the tuple

is handled similarly until reaching a singleton tuple, for which the ordering is the same as ordering the value in wrapped in the singleton tuple. The $\dot{<}$ and $\dot{\leq}$ for matrices are implemented similarly to tuples.

The implementation of $\dot{<}$ and $\dot{\leq}$ for most representations is straightforward since they can reuse the corresponding operator from the underlying representation. The following vertical rule together with appropriate `where` statements is used for the implementation of $\dot{<}$: `&a .< &b` $\leadsto$ `refn(&a) .< refn(&b)`. The rule for $\dot{\leq}$ is identical for these representations.

8.3 Avoiding conflicting symmetry breaking constraints

Conjure only breaks modelling symmetries introduced that are introduced through the refinement of an abstract domain. The symmetry is introduced into the model by Conjure and is broken immediately. Symmetry-breaking constraints are avoided in two ways. First, the users of Conjure are limited to writing constraints using operators in Essence and involving decision variables and parameters of the problem. They do not have access to the collection of decision variables that are used to concretely represent the abstract decision variables in Essence. Second, during the execution of Conjure there never exists more than one modelling symmetry in the model. A modelling symmetry is temporarily introduced and broken before any other modelling symmetry is introduced. The symmetry breaking is done completely independently for each abstract decision variable.

8.4 Summary

The technique presented in this chapter extends the rule language of Conjure by adding two new operators, and modifies structural constraints of those representation selection rules which introduce modelling symmetry. This way symmetry can be broken cheaply and automatically as it enters the model through the modelling process, increasing the quality of the models that Conjure can produce beyond model kernels.

We have shown how symmetry can be broken cheaply and automatically as it enters the model through the modelling process, increasing the quality of the models that Conjure can produce beyond model kernels.

Chapter 9

Experimental Evaluation

This chapter evaluates the operation of Conjure using three experiments. The first experiment tests the scalability of Conjure using nested abstract domains. The second experiment consists of running Conjure on a wide selection of problem specifications and investigating whether it can generate *kernels* of published CP models. The third experiment is a first iteration on model selection amongst the set of models generated by Conjure. For this experiment two different techniques are used and evaluated: racing and the Compact heuristic.

Disclaimer: Parts of this work was published in [Akg+11b] and [Akg+13b]. Authors of both papers contributed to the development of these ideas. However, I had significant contributions to both papers: in particular I designed, implemented and ran all the experiments including the racing method; designed the Compact heuristic; and implemented the required functionality in Conjure.

9.1 Scalability of Conjure

This section presents an experiment which tests the scalability of Conjure using nested abstract domains. Handling of nested domains is the best way to stress test the performance of Conjure, every level of nesting results in the creation of several concrete decision variables and the addition of new structural constraints.

In realistic Essence inputs, we typically see nesting of abstract domains up to at most 4 levels. However, Conjure is able to handle arbitrary levels of nesting, subject to time and memory constraints for the computation.

In this experiment, 4 distinct categories of Essence inputs are constructed with 11 cases in each category. The first category uses an integer domain as the base case: `int(a..b)`. In the following 10 cases, it successively adds a domain constructor of the form `set (size n) of` around the previous domain. Hence, the 11th case will have a set domain nested 10 times and containing an integer domain at the very bottom. The second category is similar but uses `maxSize n` instead of `size n`. A variable cardinality set uses boolean marker variables and hence it will generate larger output models. The third and forth categories of Essence specifications use `mset` domain constructors with a `size n` and `maxSize n` attributes respectively. In all these cases `a`, `b`, and `n` are problem parameters. The refinement will be performed at the problem class level and can later be instantiated using any value of these parameters.

Since Conjure operates at the problem class level, its performance is amortised across all the instances of the problem class at hand. Typically, users of CP technology will need to solve several instances of the same problem class yet they will only need to run Conjure once.

Table 9.1 presents the results of this experiment with a 1-hour time-limit. The reported memory figures are maximum residency; and in general around %30 of the time is spent for garbage collection. In these experiments, Conjure is able to generate output models up to 7 levels of nesting. Both its time and memory usage grow pretty fast despite the input size not growing a lot. However it is worth noting that the sizes of the output models grow very fast too. For example, without quantification unrolling and starting with no problem constraints in the input Essence, the output of 7-nested `set (size n)` case contains 2885 lines of Essence′ constraints.

Table 9.2 presents the results of another scalability experiment, in which the number of constraints in a problem specification is varied from 0 up to 100 by adding 10 constraints at

Nesting	`set (size n)`		`set (maxSize n)`		`mset (size n)`		`mset (maxSize n)`	
	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.
0	< 1s	6MB	< 1s	6MB	< 1s	6MB	< 1s	6MB
1	< 2s	6MB	< 2s	6MB	< 2s	9MB	< 2s	6MB
2	3.81s	12MB	6.76s	16MB	4.33s	11MB	9.41s	20MB
3	17.09s	32MB	44.59s	79MB	32.80s	66MB	58.71s	116MB
4	63.59s	91MB	184.55s	363MB	461.22s	668MB	673.23s	902MB
5	223.81s	337MB	670.23s	896MB	2902.23s	5482MB	Time out.	
6	631.25s	765MB	2404.60s	2870MB	Time out.		Time out.	
7	1911.24s	2393MB	Time out.		Time out.		Time out.	
8	Time out.		Time out.		Time out.		Time out.	
9	Time out.		Time out.		Time out.		Time out.	
10	Time out.		Time out.		Time out.		Time out.	

Table 9.1: Scaling with respect to levels of nesting

Number of constraints	`set (size n)`		`mset (size n)`	
	Time	Mem.	Time	Mem.
0	1.11s	7MB	1.46s	8MB
10	10.08s	49MB	18.74s	82MB
20	27.03s	136MB	49.58s	204MB
30	52.12s	269MB	99.96s	507MB
40	85.77s	427MB	152.16s	661MB
50	126.16s	660MB	225.05s	1030MB
60	170.32s	770MB	311.28s	1588MB
70	224.96s	1190MB	400.42s	2176MB
80	288.46s	1435MB	520.37s	2960MB
90	355.36s	1824MB	605.79s	3104MB
100	416.91s	2607MB	761.52s	3996MB

Table 9.2: Scaling with respect to the number of constraints

every step. Each constraint is of the form `p_i in x`, where `x` is a set or a multi-set variable and `p_i` is a problem parameter. Conjure scales much better in the number of constraints in comparison to the level of nesting in domains.

9.2 Conjure can produce kernels of good models

Conjure achieves full coverage of Essence. It has at least one variable representation rule for every abstract variable type, and horizontal and vertical expression refinement rules for all the operators defined on them. This section tests the hypothesis that the *kernel*s of constraint models written by experts can be automatically generated by refining a problem's abstract specification. For two CP models to have the same model kernel, they need to share the same viewpoint, the same representation of decision variables and the same formulation of the problem constraints. Expert models can have additional features such as implied constraints or symmetry breaking constraints but these are not considered to be in the kernel of the CP model for this evaluation.

In order to do this, we take a diverse set of 32 benchmark problems drawn from the literature and refined them with Conjure. Table 9.3 presents the results: the number of generated models, papers that contain a kernel Conjure generate and the abstract parameters and variables involved in the problem. Papers containing n kernels generated by Conjure are labelled $\times n$. Notice the variety of decision variable types involved in the benchmark problems, representing a proof that the current collection of rules, the rewrite rule language, and the Conjure system as a whole is capable of refining a variety of abstract problem specifications into concrete models.

The number of models generated for a problem specification depends on the number of representation options for the involved abstract decision variables. For instance, the *Maximum Density Still Life* contains a set decision variable whose elements are tuples and currently the system has only one variable selection rule that matches this type. Problems such as *Magic Hexagon* only contain decision variables that are concrete, so do not require refinement. We did find papers containing kernels which we are currently unable to generate, for example for *Langford's Number Problem* and *Maximum Density Still Life*. These come from complex reformulations of the problem. In each of these cases, an alternative Essence specification would allow Conjure to generate the missing kernel.

Further research is necessary to improve the quality of generated models. This is not surprising since producing a good model is well known to be difficult even by human modellers. We have established that good rewrite rules are applicable to many problems and we hope as our refinement rules database improves further, we will produce better models for all problems.

This section demonstrates Conjure's ability to reproduce the kernels of the constraint models of 32 benchmark problems found in the literature. It achieves full coverage of the Essence language via a new domain-specific rule language, whose features include: fine-grained refinement to avoid the need for flattening, which, as we have demonstrated, can impair the models produced; horizontal rules that normalise expressions to reduce considerably the total number of rules necessary for refinement; easy extensibility.

In future we of course wish to go beyond model kernels to produce full models of the same quality as those found in the literature, including symmetry breaking and implied constraints. Conjure's flexible rule-based architecture is ideally placed to achieve these aims in large part by adding new rules to those available (cf. the example in the previous subsection). Furthermore, we will prune the set of models produced to contain only the most effective models. In part, we plan to achieve this by applying a prioritisation system to rule application. This will allow refinement paths that are provably superior to dominate those shown to be weaker.

As noted, for a given specification Conjure is typically able to produce a large number of models. A principal item of future work is to reduce this number to just the best models. Conjure has begun this process by adding *precedence levels* to the refinement rules. Specifically, we introduce a multiple level structure for the horizontal and vertical refinement rules. In this setting, rules at the same level can be applied to an applicable term simultaneously. However, a rule at a lower level will always have a higher precedence than a rule at a higher level. This structure provides us with the necessary facilities to prune the set of generated models. If a rule is known to be dominant to another one, it is simply declared to be at a lower level.

For example, the refinement rules database contains horizontal refinement rules to enable the refinement of set equality constraint without giving a specific rule for that. If there is no specific rule matching with set equality for some representation, Conjure will transform it into a conjunction of two `subseteq` constraints. Then the horizontal rule for `subseteq` will be applied to transform is into a universal quantification and set membership constraint. Finally the set membership constraint will be transformed into an existential quantification and equality constraint between the set elements.

This refinement is always correct, but rarely the most efficient. The rule author is of course welcome to add a specialised refinement rule, if a better refinement for set equality can be given for a specific representation. In this case, the number of generated models will be doubled as there are two possible rewritings for the same expression.

However, if the newly added rule is defined to be at a lower level than the existing horizontal rules; it will effectively prune the set of generated models to the supposedly more efficient models.

9.2.1 Case study: Golomb Ruler

The Golomb Ruler problem[1] is the problem of finding a ruler with `n` ticks such that the differences between each pair of ticks is distinct, and the value of the maximum tick is minimised.

The Golomb Ruler problem can be very concisely specified in Essence using a set variable, an optimisation statement and a single constraint as given in Figure 9.1. The set variable models the location of ticks on the ruler. Since there cannot be two ticks at the same location, using a set to model this collection is sensible. Using a set variable gives Conjure enough information to break modelling symmetry when the Explicit representation is used as in Figure 9.2 (See lines 11 and 12 for the symmetry breaking constraint). In the Occurrence representation, the choice of using a boolean array does not introduce modelling symmetry in the first place, so symmetry breaking constraints are not necessary. The only structural

[1] CSPLib problem number 6: `http://www.csplib.org`

Table 9.3: Running Conjure on benchmark problems.

Problem name	Models	Reference	Nb. abstract params[2] and vars
Car Sequencing	128	[Gra+05]	*4 functions, 1 relation*
Template Design	16	[Pro+98]	*2 function variables, 1 mapping msets to integers*
Low Autocorellation Binary Sequences	4	[Gen+99]	*1 function*
Golomb Ruler	81	[Smi+00; Pre03]	*1 set*
All-interval series	8	[Cho+02]	*2 functions*
Vessel loading	256	[Bro98]	*9 functions, 1 mapping from a set*
Perfect Square Placement	1024	[Cam+10]	*2 functions*
Social Golfers	3	[Kiz+01; Haw+05]	*multi-set of partitions*
Progressive Party	81	[Smi+95]	*1 set, 1 set of functions*
Schur's Lemma	81	[Fle+02b]×2	*1 partition*
Traffic Lights	2	[How98]	*1 set of functions mapping integers to tuples*
Magic Squares	1	[Ref04]	*1 2-dimensional matrix*
Bus Driver Scheduling	27	[Mul98]	*1 set of sets, 1 partition*
Magic Hexagon	1	Model from CSPLib 23	*1 2-dimensional matrix*
Langford's Number Problem	32	[Hni+04]	*1 function*
Round Robin Tournament Scheduling	27	[Fri+04]	*1 relation between 2 integers and 1 set*
BIBD	16	[Pet05]	*1 relation between 2 unnamed types*
Balanced Academic Curriculum Problem	512	[Hni+02]	*2 functions, 1 relations*
Rack Configuration Problem	288	[Kiz+01]	*7 functions, 1 mapping integers to sets*
Maximum Density Still Life	1	[Smi06b]	*1 set of tuples*
Word Design for DNA Computing	16	Model from CSPLib 33	*1 set of functions*
Warehouse Location Problem	16	[Van99]	*3 functions, 1 mapping tuples to integers*
Fixed Length Error Correcting Codes	16	[Fri+03]	*2 functions, 1 mapping tuples to integers*
Steel Mill	4	[Fle+02a]	*3 functions, 1 from sets*
N-Fractions Puzzle	16	[Fri+04]	*1 function*
Steiner Triple Systems	9	[Kiz+01; Haw+05]	*1 set of sets*
N-Queens Problem	4	[Hni+04]×2	*1 function*
Peaceably Co-existing Armies of Queens	1	[Smi+04]	*1 set of tuples*
Maximum Clique Problem	81	[Reg+03]	*1 set, 1 set of sets*
Graph Colouring	4	[Hao+96; Cha+97]	*1 function*
SONET Configuration	27	[Fri+05a][1]	*1 mset of sets, 1 set of sets*
Knapsack Problem	36	[Sel09]	*2 functions, 1 set*

[1] Some models in this paper have set variables, which Conjure currently always refines.
[2] Since Conjure operates at the problem class level, problem parameters need to be refined as well as decision variables.

constraint required by the Occurrence representation is the cardinality constraint as given in Figure 9.3 (See line 11). These two representations have complementary strengths in this regard: one of them does not require a cardinality constraint but introduces symmetry, whereas the other one requires a cardinality constraint but does not introduce symmetry.

In both the Explicit and the Occurrence models the maximum element of the set is modelled using an auxiliary variable. The auxiliary variable is created using a bubble expression together with the two constraints given in lines 9 and 10 in both models. These constraints make sure `aux0` –the maximum element of the set– is greater or equal to every element in the set and it is also a member of the set. The constraints come from the definition of maximum value in a set and are implemented using Figure 6.43.

Finally the problem constraint is refined using the appropriate vertical rules for both representations. Notice how the two models contain quantified expressions with simple integer domains as a result. These are the only quantified expressions supported by Essence′. One important difference between the two models is the number of constraints after unrolling. In the Explicit model, the number of constraints corresponding to the original constraint in Essence increase linearly with respect to the number of ticks, `n`. In the Occurrence model, they increase exponentially in the number of ticks.

These models present kernels of two published models for the Golomb Ruler problem. Conjure does not only generate these two models, but it also generates several other models for this problem. The variation in models come mostly from using the Explicit representation for parts of the problem and the Occurrence representation for the remaining parts. Models using multiple representations also contain channelling constraints as described in Section 4.2.6.

9.3 Automated Model Selection

The previous section shows that Conjure can successfully refine a set of model kernels (i.e. excluding symmetry breaking and implied constraints) from a given specification, and

```
language Essence 1.3

given n : int
where n >= 0

letting bound be 2 ** n

find Ticks: set (size n) of int(0..bound)

minimising max(Ticks)

such that
  forAll pair1_1, pair1_2 in Ticks , pair1_1 < pair1_2 .
    forAll pair2_1, pair2_2 in Ticks , pair2_1 < pair2_2 .
      (pair1_1, pair1_2) != (pair2_1, pair2_2)
      ->
      max({pair1_1, pair1_2}) - min({pair1_1, pair1_2}) !=
      max({pair2_1, pair2_2}) - min({pair2_1, pair2_2})
```

Figure 9.1: The Essence specification of the Golomb Ruler problem

that this set contains the kernels of *effective* models. The addition of symmetry breaking constraints as described in Chapter 8 further enhances the quality of models that Conjure can produce. This sections presents a method of automatically selecting an effective subset of models from among the set of all models it can produce. The analysis is done for a problem class. The method takes as input a problem specification in Essence and a set of instances representative of the distribution of instances a user wishes to solve.

The subset chosen contains all those models that are not significantly outperformed across the set of supplied instances. This naturally suggests the notion of a *model portfolio*, analogous to algorithm portfolios [Hub+97; Gom+01]. No claim is made that the output set of models forms a good portfolio containing diverse models, such a claim requires more investigation.

The following methodology is used to test this hypothesis. Our measure of quality of a model with respect to an instance is the time taken for SavileRow to instantiate the model and translate for input to the Minion constraint solver plus the time taken for Minion to

```
language ESSENCE' 1.0

given n: int
where n >= 0

letting bound be 2 ** n

find Ticks_Explicit: matrix indexed by [int(1..n)] of int(0..bound)
find aux0: int(0..bound)

minimising aux0

such that
 forAll q0 : int(1..n) . Ticks_Explicit[q0] <= aux0,
 exists q0 : int(1..n) . Ticks_Explicit[q0] = aux0,
 forAll q0 : int(1..n - 1)
  . Ticks_Explicit[q0] < Ticks_Explicit[q0 + 1],
 forAll q0 : int(1..n)
  . (forAll q1 : int(1..n)
     . Ticks_Explicit[q0] < Ticks_Explicit[q1]
     ->
     (forAll q2 : int(1..n)
      . (forAll q3 : int(1..n)
         . Ticks_Explicit[q2] < Ticks_Explicit[q3]
          ->
          (Ticks_Explicit[q0] != Ticks_Explicit[q2]
         \/
         Ticks_Explicit[q1] != Ticks_Explicit[q3]
         ->
         max(Ticks_Explicit[q0], Ticks_Explicit[q1])
         -
         min(Ticks_Explicit[q0], Ticks_Explicit[q1])
         !=
         max(Ticks_Explicit[q2], Ticks_Explicit[q3])
         -
         min(Ticks_Explicit[q2], Ticks_Explicit[q3])))))
```

Figure 9.2: The Essence′ model for the Golomb Ruler problem using the `Explicit` representation

```
language ESSENCE' 1.0

given n: int
where n >= 0

letting bound be 2 ** n

find Ticks_Occurrence: matrix indexed by [int(0..bound)] of bool
find aux0: int(0..bound)

minimising aux0

such that
 forAll q0 : int(0..bound) . Ticks_Occurrence[q0] -> q0 <= aux0,
 Ticks_Occurrence[aux0],
 (sum q0 : int(0..bound) . Ticks_Occurrence[q0]) = n,
 forAll q0 : int(0..bound)
  . Ticks_Occurrence[q0]
  ->
  (forAll q1 : int(0..bound)
    . q0 < q1 /\ Ticks_Occurrence[q1]
    ->
    (forAll q2 : int(0..bound)
     . Ticks_Occurrence[q2]
     ->
     (forAll q3 : int(0..bound)
       . q2 < q3 /\ Ticks_Occurrence[q3]
        ->
        (q0 != q2 \/ q1 != q3
       ->
       max(q0, q1) - min(q0, q1) !=
       max(q2, q3) - min(q2, q3)))))
```

Figure 9.3: The Essence′ model for the Golomb Ruler problem using the `Occurrence` representation

solve the instance. The time taken by SavileRow is added since it adds instance-specific optimisations to the model, such as common subexpression elimination [Gen+08], which are desirable in practice. We iterate over the set of instances supplied with a specification, and for each we conduct a *race* [Bir+02] based on this quality measure. The 'winners' of this instance race are the set of models containing the model solved most quickly and every other model within a factor of two of that time or solved within 10 seconds (so as to prevent trivial instances from pruning). The timeout used is 1 hour. Therefore, for a particular instance, if the fastest model is solved in 30 minutes or more, the result for this instance race is the set of models entered for the race. The set of models entered into the race for instance i are the winners of the race for instance $i-1$. After we have iterated over all of the supplied instances, the subset of models remaining is selected for the specified class.

This process is predicated on the assumption that some models perform well on all instances, and therefore the order of iteration over the supplied instances is unimportant. In the experiments thus far, this assumption has held for the benchmarks and instances tested. This, however, is not expected to always to hold. For example, in a problem class whose instances vary substantially in size representational choices for the smallest instances may not be the best for the largest, and vice-versa. In future, the production of alternative models for different subdivisions of the instance space can be studied.

9.3.0.1 Heuristic Model Selection

As a contrast (and for comparison) to the racing approach described above, this section presents a very lightweight heuristic approach. The heuristic will select a model without even generating multiple complete models, and without running tests using Savile Row and Minion. It will not even require problem instances. Since it is much more lightweight and has less information to draw on, it cannot be expected to be as accurate as the racing approach. Nonetheless its performance in comparison to racing is demonstrated in the following.

The heuristic is named Compact and it is applied at each point where an abstract type

```
given d, lam, q, v: int(1..)
letting Character be domain int(1..q)
letting Index be domain int(1..lam * q)
letting String be domain function (total) Index --> Character
find E : set (size v) of String
such that   forAll s in E . forAll a : Character .
                (sum i : Index . toInt(s(i) = a)) = lam,
            forAll s1, s2 in E, s1 != s2 .
                (sum i : Index . toInt(s1(i) != s2(i))) = d
```

Figure 9.4: Essence specification of the EFPA Problem

or a constraint expression may be refined in multiple ways. For an abstract type, it defines an ordering as follows: concrete domains (such as `bool`, `matrix`) are smaller than abstract domains; within concrete domains, `bool` is smaller than `int` and `int` is smaller than `matrix`; these rules are applied recursively, so that a one-dimensional matrix of `int` is smaller than any two-dimensional matrix; abstract types also have an ordering `set < mset < function < relation < partition` and this ordering is also applied recursively. Compact will select the smallest domain according to this order. For a constraint expression (and the objective), Compact simply chooses the refinement with the least depth of the abstract syntax tree.

9.3.0.2 Case Study: Equidistant Frequency Permutation Arrays

The model selection process is demonstrated using the Equidistant Frequency Permutation Array (EFPA) problem [Huc+09]: 'The problem has parameters v, q, λ, d and it is to find a set E of size v, of sequences of length $q\lambda$, such that each sequence contains λ of each symbol in the set $\{1,\ldots,q\}$. For each pair of sequences in E, the pair are Hamming distance d apart (i.e. there are d places where the sequences disagree)'.

Again, this problem is concisely specified in Essence (see Figure 9.4) with a single abstract decision variable E and two constraints. The first ensures that each codeword must contain each symbol λ times, the second that each pair of codewords must differ in exactly d places. Conjure refines this specification into 45 models. The type of E is a fixed size set, containing a total function. The outer set is always modelled using the explicit

representation (as a vector of the inner type) and the symmetry is broken by ordering the vector with $\dot{<}$. The total function is refined in two ways: to a vector, or to a relation. In the latter case the relation is refined in four different ways, giving five representations of E in total. Subsets of these five are channelled and constraints are stated on different representations to create 45 models.

Conjure has a final compacting step that simplifies the models. For EFPA, some pairs of models become identical when compacted and in fact we have 37 unique models after this step. Of the problem classes used in this experiment only EFPA exhibited this behaviour. For model selection all 45 models are used.

For EFPA the 24 instances from Huczynska et al. [Huc+09] are used. In addition 12 easier instances were created by taking the satisfiable instances from Huczynska et al. and reducing v by one. Identifying instances by the tuple $\langle d, \lambda, q, v \rangle$, the first instance we race is $\langle 3, 7, 7, 5 \rangle$. This instance is exceptionally discriminating. The number of winners is 4, so 41 models are eliminated at this stage. Section 9.3.1 shows that not all problems converge so quickly. Second, the remaining models are raced on the instance $\langle 3, 8, 8, 6 \rangle$. This does not eliminate any models, although they are ranked in a different order. This process is continued for another 30 instances that eliminate no models. Instance $\langle 6, 4, 3, 12 \rangle$ eliminates one model, leaving three. Finally the last three instances eliminate no more models so the final winning set has three models.

All of the final set of models contain the vector representation of the total function. In addition, two of the models refine the function to a relation, then to a two-dimensional matrix of boolean variables (which is channelled with the vector). These two models differ on how one constraint is stated. The relative similarity of these three models shows that on this problem there is a clear cluster of similar winners among a more diverse set of models.

9.3.1 Experimental Evaluation

In this subsection we present the results of model selection for 4 problem classes. In Table 9.4 we report the time taken and number of final winners for EFPA, Progressive Party

Problem	Models	Final Winners	Elimination sequence	Total Wallclock Time
EFPA	45	3	45, 4, 4, 4, 4, ...	8 hours
PPP	81	6	81, 51, 28, 26, 18, ...	8 hours
SONET	27	1	27, 9, 9, 9, 9, ...	4 hours
SGP	4	2	4, 4, 4, 4, 3, ...	7 hours

Table 9.4: Model Selection with Racing

Problem (PPP), the SONET network design problem, and Error Correcting Codes (ECC). The experiments were run with 20 processes in parallel on a 32-core machine, so total CPU time would be approximately 20 times the wallclock time. The approach we have proposed for automated model selection is able to rapidly eliminate models and thus avoid repeatedly running the constraint solver on poor models that take the most time to solve. The total times spent for the races is found to be encouraging because this is an analysis that is done once for the problem class.

For EFPA the first instance discriminated extremely well, and eliminated 41 of 45 models. In contrast, PPP converges more slowly. Starting with 81 models, the winning set sizes are 51, 28, 26, 18, 18, 18, 6 etc. In this case several instances are required to weed out the poor models. SONET and SGP also converge more slowly than EFPA.

For the problem classes reported in Table 9.4, the Compact heuristic finds one of the winner models for EFPA and PPP. For SONET, Compact selects a model that is among the top four models, and is among the last to be eliminated. For SGP Compact selects a model that is eliminated at a late stage by racing. These are very promising results for a very simple heuristic.

9.3.2 Conclusions

This section has demonstrated significant progress towards the goal of automated constraint modelling. Furthermore, we have shown how, via a racing process, Conjure can select *effective* models from among those it can produce.

9.4 Summary

This chapter showed that Conjure can produce kernels of published CP models. However, it can also generate many other models and discriminating good models from bad ones is very hard. The chapter also provides two ways to find these good models among all the models generated by Conjure: racing and the Compact heuristic. These are promising results showing that Conjure is widely applicable, it can generate good models, and there are ways to automatically identify these good models. All of these aspects can be improved considerable with more research focused on each of them.

Chapter 10

Conclusion

This thesis presented a framework for automated modelling in CP: Conjure.

Automating aspects of the modelling process is crucial for making CP technology more widely useful and powerful for both novice and expert users of the technology. For novice users, using an abstract language removes the need to make several ad-hoc modelling decisions. For experts, separating problem specification and the process modelling has the potential to make modelling idioms reusable. Without automated modelling tools and a principled way to encode modelling transformations CP modelling will have to stay a challenging task that has to be repeated for every new problem, no matter if the problem shares common parts with other problems modelled very effectively by experts or not.

The main contribution of this thesis is a refinement based approach to automated modelling in CP using a collection of techniques which are demonstrated in the tool Conjure. Distinguishing features of Conjure and Essence are support for a rich collection of abstract domain constructors and arbitrarily nested types in the input language, operating at the problem class level instead of at the problem instance level, and the generation of multiple alternative models instead of a single model. Conjure also differs from existing tools by the use of a domain specific rewrite rule language and its special focus on ease of extensibility. Conjure achieves full coverage of the input language Essence and produces kernels of effective CP models fully automatically. It works without flattening input problem

specifications as a first step and uses representation independent horizontal rules to give *sensible defaults* to many expressions in Essence, it covers more of the input language using a very small number of rules.

The thesis also presented a domain-specific rule language for modelling transformations (Chapter 5). Having a rule language instead of merely encoding transformations internally to the tool enables easier maintenance of the rules database, makes it easier for CP modelling experts to author their own rules and does not require a recompilation of the main tool. Extensibility (Chapter 7) is a very valuable property for an automated modelling tool, because new ways of modelling existing problems are discovered continuously and their discovery generally requires rapid experimentation. Conjure provides a fruitful ground to study alternative ways of modelling problems.

Automated symmetry breaking in Conjure improves the models produced drastically (Chapter 8). Thanks to the highly abstract input language Essence, most of the symmetry in the problem can be viewed as modelling symmetry apparent in the domains of decision variables. Taking advantage of this, Conjure does not need to spend time and effort on *detecting* symmetry, it only need to break what it introduces. The symmetry breaking constraints introduced by Conjure are valid for the whole problem class, rather than specific instances of the problem.

Racing is presented (Chapter 9) as a principled way to find effective models for a problem class, when the user has access to a representative collection of instances for the problem class of interest. The Compact heuristic is presented (Chapter 9) as a more light-weight way of model selection: the heuristic does not require any instance data and works much faster because it does not even generate multiple alternative models, it generated one model only.

In order to evaluate Conjure's capabilities, three experiments were run (Chapter 9). The first is a scalability experiment. It demonstrates, using significantly larger problem specifications than those typically needed, that Conjure is able to produce output models for decision variables with nested domains. The second one compares models generated by Conjure to models published in the literature to analyse whether Conjure is able

to produce good models. This evaluation gives us confidence in that among the models produced, there are some good ones. The third experiment is an attempt at finding the needle in the haystack, identifying effective models among a large collection of equivalent models. The racing approach is presented as a costly but effective approach together with the Compact heuristic which is much cheaper in comparison and only works in some cases.

The Conjure system presented in this thesis provides a system capable of automatically generating several alternative constraint models with different trade-offs. The biggest limitation preventing practical use of a system like Conjure is lack of robust model selection methods in CP. The quality and quantity of alternative models generated by Conjure can also be viewed as a limitation, however this can only be improved together with better model selection techniques.

10.1 Future work

10.1.1 Automated model generation

Conjure provides good tool support for further research on automated CP modelling. It provides a powerful infrastructure which can be improved in several directions.

10.1.1.1 New representations

This thesis presented a variety of representation options for abstract domains of decision variables. However, this is only scratching the surface: Selecting the viewpoint is one of the most important modelling decisions, and more representation options should be added.

10.1.1.2 New abstract domains

Essence follows notions of discrete mathematics when defining its abstract domains. This proves to be very useful, the domains in Essence can be used in problem specifications of many problems without too much trouble. However, the addition of new abstract domains will make Essence even more useful and concise for the users. The new domains can be

application domain specific; such as tasks for the scheduling domain, stochastic variables for modelling stochastic problems, or primitives for inventory planning problems so inventory planners can express their problems in terms of shipments and orders.

10.1.1.3 Symmetry breaking

Conjure breaks all modelling symmetry it introduces. However, some problem specifications can have other kinds of symmetry in them that is not introduced by Conjure but introduces by the problem owner when formulating their problem in Essence. Detecting and breaking user symmetry and more importantly breaking it in a consistent way to breaking modelling symmetry is a challenging task, but has the potential to have rewarding returns.

10.1.1.4 Implied constraints

Similar to how Conjure can break modelling symmetry, it can automatically generate implied constraints leveraging from the high level and abstract nature of problem specifications in Essence. For example, when the problem contains two sets and a condition that one is a subset of the other, Conjure can add a constraint between the cardinalities of the two sets. Such a constraint is likely to be very useful especially when one of the sets have a known cardinality.

10.1.2 Automated model selection

The model selection problem is very important in CP, since a single problem can be modelled in several different ways, and there is a vast variation in solution performance depending on the model chosen. Now that we are able to automatically generate several models using Conjure, automating the model selection process naturally becomes the next big challenge. Equipped with a good way of differentiating between CP models, an automated modelling system can finally be very useful to both novice and expert users of CP technology.

The model selection problem in CP is very similar to the general algorithm selection problem [Ric76]. A thorough survey of the literature about this problem with a strong emphasis to combinatorial search problems and CP can be found in [Kot12].

There are several options to consider before we can attack this problem: to select a single model or a set of models; to work on problem classes or problem instances; if working on problem classes, how to analyse and explore the instance space. Since this is a very new area of research, instead of focusing on one we should try to enable multiple approaches simultaneously in Conjure. some possibilities are the following.

Post-Conjure analysis Using Conjure to generate all alternative models for a given problem, and using this set of concrete CP models as input for model selection. An advantage of this approach is the clear separation between model generation and selection. On the other hand, a major disadvantage is having to generate possibly thousands of models only to realise that most of them are not very promising early in model selection.

Mid-Conjure heuristics Using heuristics to choose promising transformations during Conjure. A first iteration of this approach presents [Akg+13b] promising results. In this work, Conjure locally selects the transformation which generates the most compact domain/expression.

In addition, hybrid approaches can also be taken; i.e., multiple heuristics can be used to generate a smaller set of alternative models, and this set can be used as input to a more generic model selection procedure.

Selecting models for problem classes vs problem instances In general, working on problem instances is easier for both automated model generation and for automated model selection. This should not be surprising as problem classes are essentially parameterised problem instances, and they *describe* a set of problems rather than a single problem. For this very reason, selecting good models for a problem class is also more

valuable: once selection is completed, findings can be used for all instances of the same class. This is also why we can afford more expensive analysis for model selection of problem classes, the cost will be amortised over all the instances.

Selecting for the whole class vs subdivisions of the instance space Selecting effective models for a problem class can be tempting. However, we know different models can be better for different instances — in an extreme example a problem class can be composed of two subproblems and a parameter value can be controlling which subproblem to actually solve. In such a case, the choice of an effective model highly depends on the value of the given parameter.

Selecting a single model vs multiple models Even if we limit ourselves to working on a single subdivision of the instance space or to a single instance, trying to select a single effective model can result in eliminating promising models prematurely due to possible shortcomings of the learning technique. In contrast, selecting a set of promising models can lead us to the notion of *model portfolios*, analogous to algorithm portfolios [Hub+97; Gom+01].

10.1.2.1 Better metrics for model comparison

Better metrics to compare models are needed. For instance level analysis, solution time is the ultimate metric, however it is potentially very expensive, we need proxies to this. For class level analysis, instance level metrics can be augmented with standard sampling and aggregation methods if a representative subset of instance data is available. Without such data we are left with class level symbolic analysis to compare models.

An important property of a model comparison metric is whether it can be used to compare partial models or not. If a metric has this property, a best-so-far model can be used as a *lower bound* and we can employ branch-and-bound to prune early during automated model generation, and only generate *good* models.

10.1.2.2 Other heuristics

Another direction is to explore better heuristics to guide Conjure and evaluate their relative performances. Simply, each such heuristic can be used independently to generate a portfolio of models. A more sophisticated method will be to use a hyper-heuristic to guide which one of the smaller heuristics should be used during model generation; dynamically switching heuristics for different parts of the model.

10.1.2.3 Feedback loop

Findings of model selection should be fed back to automated model generation in many ways. For example, if some models always seem to dominate, rules which generate those models can be put in higher precedence levels. Another kind of feedback can be when none of the generated models meet the expectations. In such a case, a modelling expert can come up with new modelling *tricks* and test the performance gains on the problem at hand. When a rule is discovered to improve the generated models, that rule should be added to the general database of Conjure so future problems can also benefit from the new rule.

Bibliography

[Akg+10a] Ozgur Akgun, Alan M Frisch, Brahim Hnich, Christopher Jefferson and Ian Miguel. 'Conjure Revisited: Towards Automated Constraint Modelling'. In: *Proceedings of the 9th International Workshop on Reformulating Constraint Satisfaction Problems*. 2010.

[Akg+10b] Ozgur Akgun, Ian Miguel and Christopher Jefferson. 'Portfolios of Constraint Models'. In: *Proc. of the SICSA PhD Conference, Edinburgh, Scotland*. 2010.

[Akg+10c] Ozgur Akgun, Ian Miguel and Christopher Jefferson. 'Refining Portfolios of Constraint Models with Conjure'. In: *CP 2010 - Principles and Practice of Constraint Programming, 16th International Conference, Doctoral Program*. 2010.

[Akg+11a] Ozgur Akgun, Ian Miguel and Christopher Jefferson. 'The Open Stacks Problem'. In: *ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011*. 2011.

[Akg+11b] Ozgur Akgun, Ian Miguel, Christopher Jefferson, Alan M Frisch and Brahim Hnich. 'Extensible Automated Constraint Modelling'. In: *AAAI 2011 - Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press 2011. AAAI Press, 2011. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/viewFile/3687/3830.

[Akg+13a] Ozgur Akgun, Alan M Frisch, Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale. 'An Automated Constraint Modelling and Solving Toolchain'. In: *ARW 2013 - 20th Automated Reasoning Workshop*. 2013.

[Akg+13b] Ozgur Akgun, Alan M Frisch, Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale. 'Automated Symmetry Breaking and Model Selection in Conjure'. In: *Proceedings of 19th International Conference on Principles and Practice of Constraint Programming {CP-2013}*. 2013.

[Akg+13c] Ozgur Akgun, Alan M Frisch, Christopher Jefferson and Ian Miguel. 'Automated Modelling and Model Selection in Constraint Programming'. In: *COSpeL: The first Workshop on Domain Specific Languages in Combinatorial Optimization*. 2013.

[Bel+11] Nicolas Beldiceanu and Helmut Simonis. 'A constraint seeker: Finding and ranking global constraints from examples'. In: *Proceedings of 17th International Conference on Principles and Practice of Constraint Programming {CP-2011}*. Springer, 2011, pp. 12–26.

[Bel+12] Nicolas Beldiceanu and Helmut Simonis. 'A Model Seeker: Extracting Global Constraint Models from Positive Examples'. In: *18th International Conference on Principles and Practice of Constraint Programming*. 2012, pp. 141–157.

[Bes+06] Christian Bessiere, Remi Coletta, Frédéric Koriche and Barry O'Sullivan. 'Acquiring Constraint Networks Using a SAT-based Version Space Algorithm.' In: *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*. Vol. 21. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2006, p. 1565.

[Bir+02] Mauro Birattari, Thomas Stützle, Luis Paquete and Klaus Varrentrapp. 'A Racing Algorithm for Configuring Metaheuristics'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2002, pp. 11–18.

[Bro+11] Christopher Brown, Huiqing Li and Simon Thompson. 'An expression processor: a case study in refactoring Haskell programs'. In: *Trends in Functional Programming*. Springer, 2011, pp. 31–49.

[Bro+13] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner and Tino Breddin. 'Paraphrasing: Generating Parallel Programs Using Refactoring'. In: *Formal Methods for Components and Objects*. Springer. 2013, pp. 237–256.

[Bro98] Kenneth N Brown. 'Loading supply vessels by forward checking and unenforced guillotine cuts'. In: *17th Workshop of the UK Planning and Scheduling SIG*. 1998.

[Bur+77] Rod M Burstall and John Darlington. 'A transformation system for developing recursive programs'. In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 44–67.

[Cad+00] M Cadoli, G Ianni, L Palopoli, A Schaerf and D Vasile. '{NP-SPEC}: An Executable Specification Language for Solving all Problems in {NP}'. In: *Computer Languages* 26 (2000), pp. 165–195. URL: http://citeseer.ist.psu.edu/71095.html.

[Cam+10] Hadrien Cambazard and Barry O'Sullivan. 'Propagating the Bin Packing Constraint Using Linear Programming'. In: *Proceedings of 16th International Conference on Principles and Practice of Constraint Programming {CP-2010}*. 2010, pp. 129–136. URL: http://dx.doi.org/10.1007/978-3-642-15396-9%5C_13.

[Cha+06] John Charnley, Simon Colton and Ian Miguel. 'Automatic Generation of Implied Constraints'. In: *Proc. of ECAI 2006*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2006, pp. 73–77. ISBN: 1-58603-642-4.

[Cha+97] Chia-Ming Chang, Chien-Ming Chen and Chung-Ta King. 'Using integer linear programming for instruction scheduling and register allocation in multi-issue processors'. In: *Computers and Mathematics with Applications* 34.9 (1997), pp. 1–14. ISSN: 0898-1221. DOI: DOI:10.1016/S0898-1221(97)00184-3. URL:

http://www.sciencedirect.com/science/article/B6TYJ-3SP5YCT-W/2/d53f7c16ef4b54505234bffb621fdf64.

[Cho+02] C Choi and J Lee. 'On the pruning behaviour of minimal combined models for permutation CSPs'. In: *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*. 2002.

[Fle+02a] Pierre Flener, Alan M Frisch, B Hnich, Z Kiziltan, Ian Miguel and Toby Walsh. 'Matrix Modelling: Exploiting Common Patterns in Constraint Programming'. In: *the International Workshop on Reformulating CSPs*. 2002, pp. 27–41.

[Fle+02b] Pierre Flener, Alan M Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson and Toby Walsh. 'Breaking Row and Column Symmetries in Matrix Models'. In: *Proceedings of 8th International Conference on Principles and Practice of Constraint Programming {CP-2002}*. 2002, pp. 462–476.

[Fle+03] Pierre Flener, Justin Pearson and Magnus Ågren. 'Introducing Esra, a Relational Language for Modelling Combinatorial Problems'. In: *LOPSTR 2003*. Ed. by Maurice Bruynooghe. Vol. 3018. Lecture Notes in Computer Science. Springer, 2003, pp. 214–232. ISBN: 3-540-22174-3.

[Fri+02] Alan M Frisch, Ian Miguel and Toby Walsh. 'CGRASS: A System for Transforming Constraint Satisfaction Problems'. In: *International Workshop on Constraint Solving and Constraint Logic Programming*. Ed. by Barry O'Sullivan. Vol. 2627. Lecture Notes in Computer Science. Springer, 2002, pp. 15–30. ISBN: 3-540-00986-8.

[Fri+03] Alan M Frisch, Christopher Jefferson and Ian Miguel. 'Constraints for Breaking More Row and Column Symmetries'. In: *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}*. Ed. by Francesca Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 318–332. ISBN: 978-3-540-20202-8. DOI: 10.1007/978-3-540-45193-8_22.

[Fri+04] Alan M Frisch, Christopher Jefferson and Ian Miguel. 'Symmetry Breaking as a Prelude to Implied Constraints: A Constraint Modelling Pattern'. In: *Proc. ECAI 2004*. 2004, pp. 171–175.

[Fri+05a] Alan M Frisch, Brahim Hnich, Ian Miguel, Barbara M Smith and Toby Walsh. 'Transforming and refining abstract constraint specifications'. In: *6th Symposium on Abstraction, Reformulation and Approximation*. Springer, 2005, pp. 76–91.

[Fri+05b] Alan M Frisch, Christopher Jefferson, Bernadette Martinez-Hernandez and Ian Miguel. 'The Rules of Constraint Modelling'. In: *Proc. of the IJCAI 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, 2005, pp. 109–116. ISBN: 0938075934. URL: http://www.ijcai.org/papers/1667.pdf.

[Fri+07] Alan M Frisch, Christopher Jefferson, Bernadette Martinez-Hernandez and Ian Miguel. 'Symmetry in the generation of constraint models'. In: *Proceedings of the International Symmetry Conference*. 2007.

[Fri+08] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández and Ian Miguel. 'Essence: A constraint language for specifying combinatorial problems'. en. In: *Constraints* 13.3 (June 2008), pp. 268–306. ISSN: 1383-7133. DOI: 10.1007/s10601-008-9047-y. URL: http://link.springer.com/article/10.1007/s10601-008-9047-y.

[Gen+06] IP Gent, Christopher Jefferson and Ian Miguel. 'Minion: A fast scalable constraint solver'. In: *ECAI* (2006), pp. 98–102. URL: http://books.google.com/books?hl=en%5C&lr=%5C&id=Eu6dg98YC4sC%5C&oi=fnd%5C&pg=PA98%5C&dq=Minion+A+fast,+scalable,+constraint+solver%5C&ots=D%5C_QyLhncEN%5C&sig=g18jBxh3dPNiBsXHi-UT%5C_CyoJQU.

[Gen+08] Ian P Gent, Ian Miguel and Andrea Rendl. 'Common Subexpression Elimination in Automated Constraint Modelling'. In: *Workshop on Modeling and Solving Problems with Constraints*. 2008, pp. 24–30.

[Gen+99] Ian P Gent and Barbara M Smith. 'Symmetry Breaking in Constraint Programming'. In: *ECAI'2000*. 1999, pp. 599–603.

[Gib+97] PB Gibbons and JA Webb. 'Some new results for the queens domination problem'. In: *Australasian Journal of Combinatorics* 15 (1997), pp. 145–160.

[Gom+01] Carla P Gomes and Bart Selman. 'Algorithm Portfolios'. In: *Artificial Intelligence* 126.1-2 (2001), pp. 43–62.

[Gra+05] M Gravel, C Gagné and W L Price. 'Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem'. In: *Journal of the Operational Research Society* 56.11 (2005), pp. 1287–1295. ISSN: 01605682. DOI: 10.2307/4102081. URL: http://dx.doi.org/10.2307/4102081.

[Ham+13] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory and Gilad Shainer. 'The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems'. In: *Formal Methods for Components and Objects*. Springer. 2013, pp. 218–236.

[Hao+96] Jin-Kao Hao and Raphaël Dorne. 'Empirical studies of heuristic local search for constraint solving'. In: *Proceedings of 3rd International Conference on Principles and Practice of Constraint Programming {CP-1996}*. 1996, pp. 194–208.

[Har01] Warwick Harvey. 'Symmetry Breaking and the Social Golfer Problem'. In: *Proc. SymCon-01: Symmetry in Constraints, co-located with {CP} 2001*. 2001, pp. 9–16.

[Haw+05] Peter Hawkins, Vitaly Lagoon and Peter J Stuckey. 'Solving Set Constraint Satisfaction Problems using {ROBDDs}'. In: *J. Artif. Intell. Res. (JAIR)* 24 (2005), pp. 109–156.

[Hni+02] Brahim Hnich, Zeynep Kiziltan and Toby Walsh. 'Modelling a Balanced Academic Curriculum Problem'. In: *CP-AI-OR-2002*. 2002, pp. 121–131.

[Hni+04] Brahim Hnich, Barbara M Smith and Toby Walsh. 'Dual Modelling of Permutation and Injection Problems'. In: *JAIR* 21 (2004), pp. 357–391.

[Hni03] Brahim Hnich. 'Thesis: Function variables for constraint programming'. In: *AI Commun* 16.2 (2003), pp. 131–132.

[Hod97] Wilfrid Hodges. *A shorter model theory*. Cambridge university press, 1997.

[How98] Walter Hower. 'Revisiting Global Constraint Satisfaction'. In: *Information Processing Letters* 66.1 (1998), pp. 41–48.

[Hub+97] Bernardo A Huberman, Rajan M Lukose and Tad Hogg. 'An Economics Approach to Hard Computational Problems'. In: *Science* 275.5296 (1997), pp. 51–54.

[Huc+09] Sophie Huczynska, Paul McKay, Ian Miguel and Peter Nightingale. 'Modelling Equidistant Frequency Permutation Arrays: An Application of Constraints to Mathematics'. In: *Proceedings of 15th International Conference on Principles and Practice of Constraint Programming {CP-2009}*. 2009, pp. 50–64.

[Jef+05] Christopher Jefferson and Alan M Frisch. 'Representations of Sets and Multisets in Constraint Programming'. In: *The Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*. Ed. by Brahim Hnich, Patrick Prosser and Barbara Smith. 2005, pp. 102–116.

[Jef+10] Christopher Jefferson, Neil CA Moore, Peter Nightingale and Karen E Petrie. 'Implementing logical connectives in constraint programming'. In: *Artificial Intelligence* 174.16 (2010), pp. 1407–1429.

[Jon+01] Simon Peyton Jones, Andrew Tolmach and Tony Hoare. 'Playing by the rules: rewriting as a practical optimisation technique in GHC'. In: *Haskell Workshop*. Vol. 1. 2001, pp. 203–233.

[Kiz+01] Zeynep Kiziltan and Braham Hnich. 'Symmetry Breaking in a Rack Configuration Problem'. In: *the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints*. 2001.

[Kon+10] Leslie De Koninck, Sebastian Brand and Peter J Stuckey. 'Data Independent Type Reduction for Zinc'. In: *Proceedings of the 9th International Workshop on Reformulating Constraint Satisfaction Problems*. 2010.

[Kot12] Lars Kotthoff. 'On algorithm selection, with an application to combinatorial search problems'. In: *20th European Conference on Artificial Intelligence* (Aug. 2012), pp. 480–485. URL: http://research-repository.st-andrews.ac.uk/handle/10023/2841.

[Lit+03] James Little, Cormac Gebruers, Derek G Bridge and Eugene C Freuder. 'Using Case-Based Reasoning to Write Constraint Programs'. In: *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}*. 2003, p. 983.

[Man+05] Toni Mancini and Marco Cadoli. 'Detecting and Breaking Symmetries by Reasoning on Problem Specifications'. In: *Abstraction, Reformulation and Approximation*. Vol. 3607. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 165–181.

[Mar+06] B Martınez-Hernández and AM Frisch. 'The automatic generation of redundant representations and channelling constraints'. In: *Constraint Modelling and ...* (May 2006), pp. 163–182. URL: www.cs.york.ac.uk/aig/constraints/AutoModel/channelling-modelling06.pdf%20http://www.cs.ucc.ie/cp06/CP06modelling.pdf%5C#page=46.

[Mar+08a] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J Stuckey, Maria Garcia de la Banda and Mark Wallace. 'The Design of the Zinc Modelling Language'. In: *Constraints 13(3)* (2008). ISSN: 1572-9354. URL: http://dx.doi.org/10.1007/s10601-008-9041-4.

[Mar+08b] Bernadette Martínez-Hernández and Bernadette Martinez-Hernandez. 'Thesis: The Systematic Generation of Channelled Models in Constraint Satisfaction'. PhD thesis. University of York, 2008.

[Mar10] Simon Marlow. 'Haskell 2010 language report'. In: *URL http://www. haskell. org/onlinereport/ ...* (2010). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.179.2870%20http://www.f4.fhtw-berlin.de/people/hansen/FHTW-AI/Lehre/2012SS/PProg/Uebungen/Uebung3/Haskell%20Languages%20Report%202010.pdf.

[Mea+11] C Mears, T Niven, M Jackson and M Wallace. 'Proving symmetries by model transformation'. In: *Principles and Practice of ...* (2011). URL: http://link.springer.com/chapter/10.1007/978-3-642-23786-7%5C_45.

[Mil+99] P Mills, Edward P K Tsang, R Williams, J Ford and J Borrett. *{EaCL} 1.5: An Easy Abstract Constraint Optimisation Programming Language*. Tech. rep. Colchester, UK: University of Essex, Dec. 1999.

[Mul98] Tobias Muller. 'Solving set partitioning problems with constraint programming'. In: *PAPPACT98*. 1998.

[Net+07] N Nethercote, Peter J Stuckey, R Becket, Sebastian Brand, Gregory J Duck and Guido Tack. 'MiniZinc: Towards a Standard {CP} Modelling Language'. In: *Proceedings of 13th International Conference on Principles and Practice of Constraint Programming {CP-2007}*. 2007, pp. 529–543.

[Pet05] Karen E Petrie. 'Constraint Programming, Search and Symmetry'. PhD thesis. University of Huddersfield, 2005.

[Pre03] S Prestwich. 'Negative Effects of Modeling Techniques on Search Performance'. In: *Annals of Operations Research* 118.1 (2003), pp. 137–150. ISSN: 0254-5330. URL: http://dx.doi.org/10.1023/A:1021809724362.

[Pro+98] Les Proll and Barbara M Smith. 'Integer Linear Programming and Constraint Programming Approaches to a Template Design Problem'. In: *INFORMS J. on Computing* 10.3 (Mar. 1998), pp. 265–275. ISSN: 1526-5528. DOI: 10.1287/ijoc.10.3.265. URL: http://portal.acm.org/citation.cfm?id=767683.768185.

[Pug04] J.-F. Puget. 'Constraint programming next challenge: Simplicity of use'. In: *Proceedings of 10th International Conference on Principles and Practice of Constraint Programming {CP-2004}*. LNCS 3258 (2004). Ed. by Mark Wallace, pp. 5–8. URL: http://link.springer.com/chapter/10.1007/978-3-540-30201-8%5C_2.

[Ref04] Philippe Refalo. 'Impact-Based Search Strategies for Constraint Programming'. In: *Proceedings of 10th International Conference on Principles and Practice of Constraint Programming {CP-2004}*. Vol. 3258. 2004, pp. 557–571.

[Reg+03] Jean-Charles Regin and Jean-Charles Régin. 'Using constraint programming to solve the maximum clique problem'. In: *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}*. 2003, pp. 634–648.

[Ren10] Andrea Rendl. 'Thesis: Effective Compilation of Constraint Models'. PhD thesis. University of St. Andrews, 2010.

[Ric76] John R Rice. 'The Algorithm Selection Problem'. In: *Advances in Computers* 15 (1976), pp. 65–118.

[Sel09] Meinolf Sellmann. 'Approximated consistency for the automatic recording constraint'. In: *Comput. Oper. Res.* 36.8 (Aug. 2009), pp. 2341–2347. ISSN: 0305-0548. DOI: 10.1016/j.cor.2008.08.009. URL: http://portal.acm.org/citation.cfm?id=1501022.1501092.

[Smi+00] Barbara M Smith, Kostas Stergiou and Toby Walsh. 'Using Auxiliary Variables and Implied Constraints to Model Non-Binary Problems'. In: *17th National Conference on AI*. AAAI Press, 2000, pp. 182–187. ISBN: 0-262-51112-6. URL: http://portal.acm.org/citation.cfm?id=647288.721272.

[Smi+04] Barbara M Smith, Karen E Petrie and Ian P Gent. 'Models and Symmetry Breaking for 'Peaceable Armies of Queens''. In: *Integration of AI and OR Techniques in CP for COP*. 2004, pp. 271–286.

[Smi+95] Barbara M Smith, Sally C Brailsford, Peter M Hubbard and H Paul Williams. 'The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared'. In: *CP '95*. 1995, pp. 36–52.

[Smi06a] Barbara M Smith. *Handbook of Constraint Programming, chapter Modelling*. 2006.

[Smi06b] BM Smith. 'A dual graph translation of a problem in 'life''. In: *Proceedings of 12th International Conference on Principles and Practice of Constraint Programming {CP-2006}* 2470 (2006), pp. 402–414. URL: http://link.springer.com/chapter/10.1007/3-540-46135-3%5C_27.

[Van+99] Pascal Van Hentenryck, Laurent Michel, Laurent Perron and J-C Régin. 'Constraint programming in OPL'. In: *Principles and Practice of Declarative Programming*. Springer, 1999, pp. 98–116.

[Van99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-72030-2.

List of Figures