Breaking the Symmetries of Indistinguishable Objects

Özgür Akgün[1[0000-0001-9519-938X]], Mun See Chang[1[0000-0003-2428-6130]], Ian P. Gent[1[0000-0002-5604-7006]], and Christopher Jefferson[2[0000-0003-2979-5989]]

[1] School of Computer Science, University of St Andrews, UK
[2] School of Science and Engineering, University of Dundee, UK
{ozgur.akgun, msc2, ian.gent}@st-andrews.ac.uk,cjefferson001@dundee.ac.uk

Abstract. Indistinguishable objects often occur when modelling problems in constraint programming, as well as in other related paradigms. They occur when objects can be viewed as being drawn from a set of unlabelled objects, and the only operation allowed on them is equality testing. For example, the golfers in the social golfer problem are indistinguishable. If we do label the golfers, then any relabelling of the golfers in one solution gives another valid solution. In this paper, we show how we can break the symmetries resulting from indistinguishable objects. We show how these symmetries induce symmetries of types built from indistinguishable objects, for example in a matrix indexed by indistinguishable objects. We then show how the resulting symmetries can be broken correctly and completely. As the method can be prohibitively expensive, we also study methods for breaking the symmetry only partially. In Essence, a high-level modelling language, indistinguishable objects are encapsulated in 'unnamed types'. We provide an implementation to automatically break symmetries of unnamed types.

Keywords: Symmetries · Modelling · Constraint programming · Automated model transformations

1 Introduction

Symmetries have long been understood to be both widely occurring and a source of inefficiency for solving technologies. As a result, this has been an exceptionally well-studied topic in constraint programming [13], Boolean satisfiability [25], and mixed-integer programming [17]. A particularly important case of symmetries is where the problem has indistinguishable objects. These are objects which, when interchanged, give us essentially the same situation. For example, two machines of the same model are equivalent in a factory scheduling problem, and any valid schedule will give an equivalent schedule when two such machines are interchanged. Further complications are introduced when we have multiple sets of indistinguishable objects, and we are not allowed to interchange objects of the different sets.

When modelling problems with symmetries, due to the limited choices of representations, one tends to *introduce* symmetries that are not in the original problem. These symmetries must then be broken, e.g. by adding symmetry breaking constraints. High-level modelling languages such as ESSENCE allow one to specify the problems in a more abstract way, and symmetries introduced by modelling can then be handled automatically by CONJURE, an automatic model rewriting tool. In ESSENCE, 'unnamed types' were introduced to capture the notion of indistinguishable objects [5], to express sets of objects whose labels are not important. Compound types can be constructed in terms of these unnamed types, for example, we can have sets of tuples of indistinguishable objects. However, while unnamed types were present in the first version of ESSENCE, previously CONJURE ignored the symmetry of unnamed types and simply transformed them into integers. Handling these symmetries is significantly more difficult than other symmetries already managed by CONJURE. We show how these symmetries can be broken *automatically*, without requiring expertise in symmetry breaking.

To do so, we extend ESSENCE to support permutations of types. These permutations allow us to represent the symmetries of indistinguishable objects and provide a base for a general and extensible framework for breaking the symmetries introduced by these indistinguishable objects. By showing how we implement permutations, we show how other technologies that want to deal with the symmetries of indistinguishable objects can adopt a similar approach. As the types of ESSENCE variables can be arbitrary nested, and the list of objects we want to handle might be extended in the future, we give the semantics of ESSENCE types recursively, in terms of a much smaller set of mathematical objects. We use this semantics to define how symmetries of indistinguishable objects induce symmetries of objects constructed from them. We also show how a well-defined total ordering on a compound type can be built up in terms of the ordering of its constituent types.

These ingredients let us generalise the lex-leader symmetry breaking method to compound types. We will illustrate our technique in the context of CONJURE, but we will also discuss how other modelling languages can take the same approach to remove symmetries due to indistinguishable objects. Often we do not want to break all of the symmetries in a model since that encoding would require too many constraints and so will be detrimental to performance. In fact, checking if an assignment satisfies lex-leader symmetry breaking constraints is NP-hard in general [7]. For this reason, we also explore weaker, partial forms of symmetry breaking, offering a modelling choice between fast and complete symmetry breaking. We show, using well-known constraint models containing unnamed types, how our symmetry breaking encodings can be applied to them using this abstraction. As an example, we show that the commonly used 'double-lex' method [9] naturally arises from our methods.

In Section 2, we give a brief overview of symmetry breaking in constraint programming and the ESSENCE language, and then define the symmetries of unnamed types in Section 3. We then define the symmetries induced by unnamed types and see how we can break them, completely or incompletely, using a newly defined total order of the values of any type. In Section 4, we describe the method implemented in CONJURE and in Section 5 give some case studies.

Motivating Example: Social Golfers Problem The problem asks for a schedule for p people playing golf over w weeks in g groups per week [14]. To attain maximum socialisation, no two different golfers play in the same group as each other in two different weeks. Notice that the problem only cares whether or not two golfers are the same person or different people. That is, the golfers are all *indistinguishable*. Similarly, the groups are also indistinguishable, and so are the weeks. For example, the constraints take no account of weeks being consecutive or otherwise: two weeks are either identical or distinct. This means that given any solution to the problem, we can obtain another solution through any permutation of the golfers. Similarly, we can also permute the groups and the weeks freely. Furthermore, we can apply any permutation of the golfers, groups and weeks concurrently to get another solution. With so many symmetries in hand, manually breaking them may require some modelling expertise, whether the aim is to break all symmetries completely or to do some form of partial symmetry breaking. The approach we take in this paper allows this to be done automatically. We will do this in the context of CONJURE, working on the *unnamed types* allowed by ESSENCE, in this case for golfers, weeks and groups. Our approach is general and could be applied in other modelling languages.

2 Background

A *constraint satisfaction problem* (CSP) $\mathcal{P}$ with n variables is a triple (V, D, C), where $V = \{V_1, V_2, \ldots, V_n\}$ is the set of *variables*, D consists of sets $\mathrm{Dom}(V_i)$, called the *domain* of V_i, for each $1 \leqslant i \leqslant n$, and $C = \{C_1, C_2, \ldots, C_k\}$ is the set of *constraints*, where each C_i is a subset of the Cartesian product $\bigtimes_{1 \leqslant i \leqslant n} \mathrm{Dom}(V_i)$. An *assignment* of the variables in V is an n-tuple $(a_1, a_2, \ldots, a_n)$, where each $a_i \in \mathrm{Dom}(V_i)$. An assignment is a *solution* to $\mathcal{P}$ if it is in the intersection $\bigcap_{1 \leqslant i \leqslant k} C_i$. The *solution set* to $\mathcal{P}$ is the set of all solutions to $\mathcal{P}$.

A *permutation* of a set Ω is a bijection from Ω to itself. We typically denote permutations using the cycle notation. That is, a permutation $\sigma := (a_{11}, a_{12}, \ldots, a_{1k_1})(a_{21}, a_{22}, \ldots, a_{2k_2}) \ldots (a_{r1}, a_{r2}, \ldots, a_{1k_r})$ means that, for all i, we have $a_{ij} \mapsto a_{i(j+1)}$ for $j < k_i$ and $a_{ik_i} \mapsto a_{i1}$. The composition of two permutations σ_1 and σ_2 is denoted by $\sigma_1\sigma_2$, the inverse of a permutation σ is denoted by σ^{-1}. For $\omega \in \Omega$ and a permutation σ over Ω, we denote the image of ω under σ by ω^σ. A *permutation group* G over a set Ω is a set of permutations over Ω that is closed under compositions and inverses. Such G necessarily contains the identity 1_G, the permutation that fixes all points in Ω. The set of all permutations over Ω is called the *symmetric group* of Ω, denoted by $\mathrm{Sym}(\Omega)$.

A *group action* of a group G on a set Ω is a map $\phi : G \times \Omega \to \Omega$ such that $\phi(1_G, \omega) = \omega$ for all $\omega \in \Omega$ and $\phi(gh, \omega) = \phi(h, \phi(g, \omega))$. When such a group action exists, we say that G *acts on* Ω and G is a *symmetry group* of Ω. Further, we write ω^g for $\phi(g, \omega)$, which aligns with the notation of permutation application because $\mathrm{Sym}(\Omega)$ naturally acts on Ω. Group actions formalise what we intuitively understand as symmetries: the identity permutation should fix the object in question, and applying two permutations consecutively should be the same as applying the composition of the two permutations.

When we have a symmetry group G acting on the domains of the decision variables V, we have an equivalence relation on the domain set D (and hence the set of all solutions), where two assignments a and a' are equivalent if there exists a permutation $g \in G$ such that $a^g = a'$. A *symmetry breaking constraint* is a constraint which, when added, removes or reduces symmetric values from consideration. A *sound* (resp. *complete*) symmetry breaking is one where at least one (resp. exactly one) solution from each equivalence class is preserved. Many symmetry breaking strategies use some ordering. For a total ordering $\leqslant$ on set A, the *lexicographical ordering* $\leqslant_{lex}$ *over* $\leqslant$ is the total ordering on the tuples/matrices over A such that $(t_1, t_2, \ldots, t_k) \leqslant_{lex} (t'_1, t'_2, \ldots, t'_k)$ if and only if either $t_i = t'_i$ for all i, or there is an i such that $t_i < t'_i$ and $t_j = t'_j$ for all $j < i$.

Symmetry breaking for constraint programming is very well studied (see [13] for an overview). Two closely related concepts are interchangeability of values [10] and intensional permutability of variables [23], but they only concern value and variable symmetries respectively. Our work differs from these as we take a type-directed view of indistinguishable objects, which means that types with interchangeable values can be used to build higher-level types. Depending on how we build these higher-level types, the symmetries can be variable or value symmetries, or indeed both or neither, as we shall see. This suggests that a new, more general, method of reasoning with symmetries is needed.

Symmetries are also introduced by Conjure when the abstract types in Essence are refined to lower-level types in Essence Prime (see Section 2.1). Currently all but the symmetries arising from unnamed types are automatically broken by Conjure (see [2] for more details). Symmetries in the constraint modelling language MiniZinc is also extensively studied (see, for example, [6,18]), but our work here differs in that we consider higher-level abstract types. Outwith constraint programming, the symmetries of indistinguishable objects are exploited in the SAT solver SymChaff, which requires a symmetry description as input [24]. It is shown that such a description can be generated from annotating PDDL models, but we operate on a much higher level language. The field of lifted inference started by Poole [21] also uses the symmetries of indistinguishable objects to improve efficiency, but the focus is on probabilistic reasoning and model counting.

2.1 Essence as a Modelling Language

Essence and Essence Prime are both constraint specification languages. The domains of decision variables in an Essence or Essence Prime problem specification are defined by adding attributes and/or bounds to built-in types. For example, in Essence, we can have a variable of domain `set (size 3) of matrix indexed by int(1..5) of bool`. In this paper, matrices indexed by $[I_1, I_2, \ldots, I_k]$ refers to k-dimensional matrices where the values are accessed by values of $I_1 \times I_2 \times \ldots \times I_k$, so an entry of such a matrix m is $m[i_1, i_2, \ldots, i_k]$, where each i_j is in I_j.

Types in ESSENCE and ESSENCE PRIME are divided into two kinds: atomic and compound. The atomic types of ESSENCE PRIME are Booleans and integers, while ESSENCE further supports enumerated types and unnamed types. Compound types, such as matrices, are defined using atomic types and can be arbitrarily nested. ESSENCE PRIME only support the matrix compound domain, while ESSENCE also supports tuples, records and variants. ESSENCE further supports abstract decision variables of set, multiset, sequence, function, relation and partition. Non-abstract domains are also called *concrete* domains. For more details, see [2] or the documentation at `https://conjure.readthedocs.io`.

Remark 1. For a type T, we denote its set of all possible values by $\mathrm{Val}(T)$, which is defined in terms of matrices, multisets and tuples: the values of `bool`, `int` and `enum` are what one would expect; the values of a `tuple`, `record`, `variant` and `sequence` can be naturally defined as tuples; the values of a `matrix` are matrices, the values of a `set`, `mset`, `partition` can be naturally defined as (nested) multisets; for types τ_i, the values of a `function` $\tau_1 \rightarrow \tau_2$ are subsets of $\mathrm{Val}(\tau_1) \times \mathrm{Val}(\tau_2)$ such that there are no two elements with the same value in its first position; the values of a `relation` $(\tau_1 * \tau_2 * \ldots * \tau_k)$ are subsets of $\mathrm{Val}(\tau_1) \times \mathrm{Val}(\tau_2) \times \ldots \times \mathrm{Val}(\tau_k)$. Note that the representations presented here give abstract meaning to the types, carefully selected to simplify the theory, but may not adhere to the representation of the underlying implementations.

There are two advantages in defining $\mathrm{Val}(T)$ in terms of matrices, multisets and tuples. Firstly, this allows us to avoid large case splits over types, e.g. when defining the symmetries of compound objects built from unnamed types. Furthermore, this makes our method more general and applicable across other platforms. As long as we define the values of a type in a similar way, either for a new type in ESSENCE or a type in other systems, the method described in this paper still applies. Note also that we can also have all types to be defined as multisets and tuples only. This is because a matrix m indexed by $[\tau_1, \tau_2, \ldots, \tau_k]$ of τ can be represented in terms of multisets and tuples as $\{(i_1, i_2, \ldots, i_k, m[i_1, i_2, \ldots, i_k]) \mid i_j \in \tau_j \text{ for all } 1 \leqslant j \leqslant k\}$.

CONJURE transforms a problem specification (a *model*) in the ESSENCE language into a problem specification in the ESSENCE PRIME language, through a series of rewrites or transformations (see [2]). Concrete domains are represented directly in ESSENCE PRIME, possibly by separating into their components. These transformations are straightforward and do not introduce any symmetries.

The abstract types are removed in a series of rewrites called *refinements*. For each abstract domain, there is a choice as to how it can be translated into concrete domains. Such a choice is what we call a *representation* of the abstract domain. The constraints involving the abstract domains are rewritten according to the selected representation. In some cases, an abstract type is represented as a concrete type that satisfies certain constraints (e.g. sets as lists with all different elements). Such constraints are called *structural constraints*. We will not detail representations used in Conjure here, but will describe the ones which we use in examples. We direct interested readers to [2, Table 5] for a summary of the representations in Conjure. *Modelling symmetries* are introduced if we translate abstract decision variables to a variable with a bigger domain, which may result in the increase of solution number. In Conjure, modelling symmetries without unnamed types are *always* broken completely. The complete symmetry breaking constraint for the refinement of each abstract type in Essence is well studied. Please refer to [2] for details.

3 Unnamed Types and How to Break Them

To model indistinguishable objects, we use the concept from Essence of 'unnamed types'. While Essence provides unnamed types as a built-in type and we implement our techniques in Conjure, our work applies generally. For any modelling situation to which unnamed types apply, the techniques we propose can be used whether or not the modelling language used provides unnamed types. The advantage of having unnamed types built in, as in Essence, is simply that no additional work is necessary to recognise the existence of indistinguishable objects. In this section we show the value in modelling with unnamed types, discuss the symmetries inherent in unnamed types and how to break them.

3.1 Modelling with Unnamed Types

We shall briefly show some examples of how one would use unnamed types in modelling, to illustrate their usefulness for high-level modelling and the difficulties in breaking their symmetries. Note that we can model these problems using more abstract Essence types, but we choose to avoid these abstract types in this paper in hope to better illustrate the potential use of unnamed types.

Recall the **social golfer problem** from Section 1. A model may have, as decision variable, `matrix indexed by [int(1..w), int(1..p)] of int(1..g)`, together with constraints for maximal socialisation and to make sure that the group sizes are as expected, which we omit here. As we have seen, the golfers, weeks and groups can be permuted while still giving us valid schedules. The labels for golfers, weeks and groups, encoded as integers here, do not matter – permutations of them, when done consistently, will give us equivalent solutions. So we can define `golfers`, `weeks` and `groups` as unnamed types of size p, w and g respectively, using the syntax `letting golfers be new type of size p`, and similarly for `weeks` and `groups`. Then we can take the decision variable to be `matrix indexed by [weeks, golfers] of groups`, and let Conjure handle the symmetries of unnamed types automatically.

The **template design problem** [27] arises in a printing factory that is asked to print $c_1, c_2, \ldots, c_k$ copies of designs $d_1, d_2, \ldots, d_k$ respectively. Designs are printed on large sheets of paper and each sheet can hold at most s designs. A *template* is defined by the designs to be printed on a sheet (at most s of them, can be repeated, order does not matter). Given a number n, we want to find n

templates $t_1, t_2, \ldots, t_n$, and the number of copies for each of them to satisfy the printing order, while minimising the total number of printing. One might model this problem with two decision variables: (i) `matrix indexed by [int(1..n)] of int`, called M_1, to encode the number of copies needed for each template; and (ii) `matrix indexed by [int(1..n), int(1..k)] of int(1..s)`, called M_2, to encode the number of copies of each design in each template. As before, the labels of the templates, currently integers 1 to n, do not matter – permuting the labels gives equivalent solutions. Since the templates are used as indices in two decision variables, any symmetry handling of one must be consistent with the other. For example, enforcing that M_1 must be sorted and at the same time enforcing that rows of M_2 are sorted may give us a wrong result. An alternative would be to replace `int(1..n)` in the indices for both M_1 and M_2 with an unnamed type of size n. Conjure shall then automatically and consistently break their symmetries.

The **set-theoretic Yang-Baxter problem** asks for a special class of solutions to the infamous Yang-Baxter equations, which gives insights to various subfields of algebra and combinatorics (see [3] for references). A special class of the set-theoretic solutions of the Yang-Baxter equation can be modelled as a mapping $\varphi : X \times X \to X$ which satisfies certain constraints (see [3] for details), for a set X. As is common in mathematics, the elements X are unlabelled and interchangeable. In this case, if X is realised as a concrete set of $\{x_1, x_2, \ldots, x_n\}$, then any permutation of the elements of X in the definition of φ gives another mapping that is essentially the same (or equivalent). As, again, is common in mathematics, we want to count the number of solutions up to equivalences, which means that we want to remove the symmetries due to the interchangeability of the elements of X. Such a map φ can therefore be modelled as a `matrix indexed by [T,T] of T`, where T is an unnamed type. In this problem, the same unnamed type is used both as indices (twice) and elements of a matrix, so swapping two values in T requires swapping two rows and two columns, and also all occurrences of the two values for all variables. So the symmetries of this matrix is neither a variable nor a value symmetry (see [13]), the two most well-studied families of symmetries in constraint programming, as we require the synchronisation of both. Expressing correct symmetry breaking constraints requires significant expertise in constraint modelling, which limits the ability of many constraint users to deal with the symmetries of their problems.

3.2 Symmetries of Unnamed Types

As we have seen above, when modelling, we often want to express that two items are equivalent or indistinguishable from each other. In Essence, we model these using *unnamed types*, which are sets of known size with implicit symmetries: values of an unnamed type are unlabelled and hence interchangeable. The values of unnamed types are not ordered and the only operations allowed on unnamed types are equality and inequality. Unnamed types are atomic so they can be used to construct compound domains in many ways, including as members of a set, the domain or image of a function, or the indices of a matrix.

In order to define symmetries inhabited by variables constructed from unnamed types, we need to first define the set of possible values of unnamed types. This proves to be difficult since as soon as we enumerate its values, we have put a label on its elements and hence introduced symmetries. We therefore define unnamed types as an enumerated type that comes with a symmetry group:

Definition 1. *An* unnamed type T of size n *is an* Essence *type with value set* $\mathrm{Val}(T) = \{1_T, 2_T, \ldots, n_T\}$*, together with a symmetry group* $\mathrm{Sym}(\mathrm{Val}(T))$ *consisting of all permutations of* $\mathrm{Val}(T)$*.*

For example, in the social golfer problem, we can represent three possible golfers as an unnamed type G of size 3. Then $\mathrm{Val}(G)$ is $\{1_G, 2_G, 3_G\}$. Note that a value of an unnamed type is unique to its type. For example, 1_T can only be a value of the unnamed type T, and not at the same time be a value of another unnamed type U. This means that the values of distinct unnamed types T and U are always disjoint. To simplify notations, we write $\mathrm{Sym}(T)$ for $\mathrm{Sym}(\mathrm{Val}(T))$.

As we are dealing with indistinguishable objects, the symmetry group in Definition 1 is a symmetric group. However, our method of breaking these symmetries completely never uses the properties of symmetric groups. So the method is generalisable to the case where we have an atomic type T with any permutation group G on its values as its symmetry group.

Induced Symmetries on Compound Types When solving a problem with unnamed types, we only want to retain the solutions up to symmetries. In the social golfer example, a solution is equivalent to another assignment where we have swapped two of the golfers, and so we should only output one of them. However, a decision variable can be an arbitrarily-nested construction of various Essence types. So next we define how the symmetries of unnamed types induce symmetries of the compound variables constructed from unnamed types.

Definition 2. *Let T be an unnamed type of size n and X be a compound type. Then* $\mathrm{Sym}(T)$ *is a symmetry group of* $\mathrm{Val}(X)$*, where the action is defined recursively by: for all $g \in \mathrm{Sym}(T)$, and $x \in \mathrm{Val}(X)$,*

1. *if x is a value of type T, then x^g is the image under the action on* $\mathrm{Val}(T)$*;*
2. *if x is atomic and x is not of type T, then $x^g = x$;*
3. *if x is a matrix indexed by $[I_1, I_2, \ldots, I_k]$ of E, then the image x^g is a matrix where its i-th element $x^g[i]$ is $(x[i^{(g^{-1})}])^g$, where $i^{(g^{-1})}$ denotes the preimage of i under g;*
4. *if x is a multiset $\{v_1, v_2, \ldots, v_k\}$, then $x^g = \{v_1^g, v_2^g, \ldots, v_k^g\}$;*
5. *if x is a tuple $(v_1, v_2, \ldots, v_k)$, then $x^g = (v_1^g, v_2^g, \ldots, v_k^g)$.*

One can check that this indeed gives a group action. While there may be other possible group actions, we chose the most natural one. Recall from Remark 1 that possible values of a non-atomic variable can be constructed from only matrices, multisets and tuples. So Definition 2 *does* in fact define the image of all possible types in Essence, by deducing from Remark 1, and considering sets as a special case of multisets (with multiplicity one for each element). As noted earlier, as long as we define types in a similar way, we can obtain the action on any compound type of other modelling languages using Definition 2.

Example 1. Let T be an unnamed type of size 3, and let X be of type `function T → int(4..5)`. A possible value of X is $x = \{(1_T, 4), (2_T, 5), (3_T, 4)\}$, representing the function that maps 1_T and 3_T to 4, and 2_T to 5. Consider the permutation $g = (1_T, 2_T)$ swapping 1_T and 2_T and leaving 3_T fixed. Then the image of x under g is $x^g = \{(1_T, 4), (2_T, 5), (3_T, 4)\}^g = \{(1_T, 4)^g, (2_T, 5)^g, (3_T, 4)^g\} = \{(1_T^g, 4^g), (2_T^g, 5^g), (3_T^g, 4^g)\} = \{(1_T^g, 4), (2_T^g, 5), (3_T^g, 4)\} = \{(2_T, 4), (1_T, 5), (3_T, 4)\}$, representing a function that maps 2_T and 3_T to 4, and 1_T to 5, and where the subsequent expression rewriting uses Parts 4,5,2,1 of Definition 2 respectively.

One may find the presence of preimage in the matrix index of Definition 2 to be unintuitive, but it is needed so we have a group action. One can also check that this definition is consistent to if we represent matrices as sets of tuples.

Example 2. Consider a 1-dimensional matrix $m = [a, b, c]$ indexed by elements of unnamed types $1_T, 2_T, 3_T$. Let g be the permutation $(1_T, 2_T, 3_T)$ and h be $(1_T, 2_T)$. From Definition 2, we find that the image of m first under g and then under h is $(m^g)^h = [c, a, b]^h = [a, c, b]$. We get the same value if we take the image of m under the composition of g and h, as $gh = (2_T, 3_T)$ and $m^{gh} = [a, b, c]^{(2_T, 3_T)} = [a, c, b]$. However, if Definition 2 had defined $(m^g)[i]$ to be $(m[i^g])^g$ instead, $(m^g)^h = [b, c, a]^h = [c, b, a]$, which is not m^{gh}.

Symmetries of Multiple Unnamed Types If a variable X is constructed from multiple unnamed types, say T and U, any combination of elements in $\mathrm{Sym}(T)$ and $\mathrm{Sym}(U)$ also permute $\mathrm{Val}(X)$. We say that the direct product $\mathrm{Sym}(T) \times \mathrm{Sym}(U)$ also acts on $\mathrm{Val}(X)$. In general, the *direct product* of groups $G_1, G_2, \ldots, G_k$ is the Cartesian product $G_1 \times G_2 \times \ldots \times G_k$ consisting of all k-tuple $(g_1, g_2, \ldots, g_k)$ where each $g_i \in G_i$. If each G_i acts on a set Ω_i and the Ω_i's are disjoint, the direct product $G_1 \times G_2 \times \ldots \times G_k$ acts on the disjoint union $\bigcup_{1 \leqslant i \leqslant k} \Omega_i$ by $\alpha^{(g_1, g_2, \ldots, g_k)} = \alpha^{g_i}$ if $\alpha \in \Omega_i$.

Definition 3. *Let $T_1, T_2, \ldots, T_m$ be distinct unnamed types. Then the direct product $D := \mathrm{Sym}(T_1) \times \mathrm{Sym}(T_2) \times \ldots \times \mathrm{Sym}(T_m)$ is a symmetry group of $\mathrm{Val}(X)$, where the action is defined by $v^{(g_1, g_2, \ldots, g_m)} = (\cdots((v^{g_1})^{g_2})^{\cdots})^{g_m}$, for each element $(g_1, g_2, \ldots, g_m)$ of D and $v \in \mathrm{Val}(X)$, and the application of each g_i is as defined in Definition 2.*

This gives a group action because each $\mathrm{Sym}(T_i)$ acts on $\mathrm{Val}(T_i)$ and distinct unnamed types are disjoint sets. Further, as the g_i's permute disjoint sets of points, they commute. That is, $g_i g_j = g_j g_i$ for all i, j. Then, since we have a group action, taking images under them is commutative since $(v^{g_i})^{g_j} = v^{(g_i g_j)} = v^{(g_j g_i)} = (v^{g_j})^{g_i}$ for all i, j. So the order in which we take the images when considering permutations of different unnamed types does not matter, hence the order of unnamed types in the direct product also does not matter.

Example 3. Let T and U be unnamed types of size 2 and 4 respectively, and let M be of type `matrix indexed by [`T`, int(1..3)] of` U. Then $D = \mathrm{Sym}(T) \times \mathrm{Sym}(U)$ acts on $\mathrm{Val}(M)$. Consider $m = [[1_U, 2_U, 3_U], [2_U, 3_U, 4_U]] \in \mathrm{Val}(M)$ where we write the elements of m in the order $1_T, 2_T$, and let $g = (1_T, 2_T)$ and $h = (1_U, 3_U)(2_U, 4_U)$. Here g swaps 1_T and 2_T, whereas h and swaps 1_U and 3_U and also 2_U and 4_U at the same time. Then $(g, h) \in D$ and by Definition 3, $m^{(g,h)} = ([[1_U, 2_U, 3_U], [2_U, 3_U, 4_U]]^g)^h$, which gives $[[2_U, 3_U, 4_U]^g$, $[1_U, 2_U, 3_U]^g]^h = [[2_U, 3_U, 4_U], [1_U, 2_U, 3_U]]^h$ as g permutes the indices and fixes values not from T. Now as h fixes the indices $1_T, 2_T$ and the indices $1, 2, 3$, this is just $[[2_U, 3_U, 4_U]^h, [1_U, 2_U, 3_U]^h] = [[(2_U)^h, (3_U)^h, (4_U)^h], [(1_U)^h, (2_U)^h, (3_U)^h]]$. Finally, h permutes values in $\mathrm{Val}(U)$, so $m^g = [[4_U, 1_U, 2_U], [3_U, 4_U, 1_U]]$.

If X is our only decision variable, then we obtain an equivalence relation on the set of all solutions, where two solutions x and y in $\mathrm{Val}(X)$ are equivalent if $x^g = y$ for some $g \in G$. If we have multiple decision variables $V_1, V_2, \ldots, V_d$, then we can reduce to the case where there is only one decision variable, which is the tuple $(V_1, V_2, \ldots, V_d)$. This is particularly important when we want to ensure the consistent application of permutations of unnamed types across multiple variables, such as in the template design problem from Section 3.1.

3.3 Breaking the Symmetries of Unnamed Types

A common and general way to break symmetries is to use the lex-leader constraints [7]. In general, for a group G acting on the domain $\mathrm{Val}(X)$ of a variable X, the (value) lex-leader constraint $LL_{\preceq}(G, X)$ for X under G with respect to a total ordering $\preceq$ of $\mathrm{Val}(X)$ is the constraint $\forall \sigma \in G.\, X \preceq X^{\sigma}$. This asserts that, in any G-induced equivalence class of $\mathrm{Val}(X)$, only one value (the smallest one) can be assigned to X. Recall that we treat our problems as containing a single variable, which may be a tuple, so we need to only consider symmetry breaking for a single variable X. In this paper, we always base our symmetry breaking on a lex-leader constraint $LL_{\preceq}(G, X)$. In this situation, sound symmetry breaking constraints will be implied by $LL_{\preceq}(G, X)$, and complete symmetry breaking constraints will imply $LL_{\preceq}(G, X)$ as well.

To completely break the unnamed type symmetries of a variable X, we use $LL_{\preceq}(G, X)$, where G is the symmetry group acting on $\mathrm{Val}(X)$ and $\preceq$ is a total ordering on $\mathrm{Val}(X)$. So we can eliminate unnamed type symmetries in the following way, the proof of which follows from the correctness of lex-leader constraints eliminating all but one solution in each equivalence class.

Proposition 1. *Starting with an* Essence *model* M *with unnamed types* $T_1, T_2, \ldots, T_k$ *of size* $s^1, s^2, \ldots, s^k$ *respectively, we can obtain an equivalent model (in the sense that there is a bijection between the solution sets)* without *unnamed types in the following way. Letting* $V_1, V_2, \ldots, V_n$ *be the decision variables of* M*, first replace each unnamed type* T_i *by an enumerated type with values* $1_{T_i}, 2_{T_i}, \ldots, s^i_{T_i}$*. Then let* X *be a new decision variable representing the tuple* $(V_1, V_2, \ldots, V_n)$*. Finally, we add the lex-leader constraint* $LL_{\preceq}(\mathrm{Sym}(T_1) \times \mathrm{Sym}(T_2) \times \ldots \times \mathrm{Sym}(T_k), X)$*, where* $\preceq$ *is a total ordering on* $\mathrm{Val}(X)$*.*

Therefore, we need a total ordering of $\mathrm{Val}(T)$ for any type T that is not constructed from unnamed types. We shall define these orderings in the next subsection. This will then inform us on how we can refine constraints of the form $X \preceq Y$ when X and Y are of abstract types.

Before we move on, note that there are many papers which consider different methods of generating subsets of symmetry breaking constraints (see [13] for an overview). The most common technique is to replace $LL_{\preceq}(G, X)$ with $LL_{\preceq}(S, X)$ for some subset S of G, to obtain sound but incomplete symmetry breaking constraints. When G is a direct product $G_1 \times G_2$, we may use $LL_{\preceq}(G_1 \cup G_2, X)$ instead. When G is the symmetric group $\mathrm{Sym}(\{x_1, x_2, \ldots, x_n\})$, we can take $LL_{\preceq}(S, X)$ where $S = \{(x_i, x_j) \mid 1 \leqslant i, j \leqslant n\}$ consists of all permutations in G that swap two elements. Alternatively, we can take S to be $\{(x_i, x_{i+1}) \mid 1 \leqslant i < n\}$, the set of all permutations in G that swaps any consecutive points. Examples of the constraints added can be found in the .trace files of our repository `https://github.com/stacs-cp/CPAIOR2025-Symmetry`.

Total Ordering for All Types We shall describe a general approach to defining a total ordering $\preceq_T$ of $\mathrm{Val}(T)$ for any given type T not constructed from unnamed types. To simplify notations, we drop the subscript T when doing so will not cause confusion. The actual ordering used does not matter for correctness, as long as it is a total order. We first define an order on multisets in terms of the ordering on its members' type. We then show how ordering on other types can be defined in terms of the multiset ordering. This is the ordering used in our implementation in CONJURE.

For the ordering on multiset, we used an ordering very similar to one in the literature [12,16]. Let M be a type consisting of multisets of elements of type S and $\preceq_S$ is an ordering of $\mathrm{Val}(S)$. We say that $m_1 \preceq_M m_2$ if and only if one of the following is true: (i) $m_2 = \varnothing$; (ii) $m_1, m_2 \neq \varnothing$ and $\min(m_1) \prec_S \min(m_2)$; (iii) $m_1, m_2 \neq \varnothing$ and $\min(m_1) = \min(m_2)$ and $m_1 \backslash \{\min(m_1)\} \preceq_M m_2 \backslash \{\min(m_2)\}$. This ordering may be unintuitive, but it is chosen so that ordering on multisets is equivalent to lex-ordering of a natural representation (specifically the occurrence representation; see, for example, [16] for proof). Since multisets are abstract types, constraints $X \preceq_M Y$, when X and Y are multisets, will need to be refined, and we can do so using this occurrence representation. The ordering for all other types can be found in the following definition.

Definition 4. *Let T be an* ESSENCE *type not constructed from any unnamed types. We define a total ordering $\preceq_T$ for values $\mathrm{Val}(T)$ of type T recursively by:*

1. *if $\mathrm{Val}(T)$ consists of integers, we take $\preceq_T$ to be $\leqslant$ on integers;*
2. *if $\mathrm{Val}(T)$ consists of Boolean, we use `false` $\preceq_T$ `true`;*
3. *if $\mathrm{Val}(T)$ consists of enumerated types, then $x \preceq_T y$ if x occurs before y in the definition of the enumerated type;*
4. *if $\mathrm{Val}(T)$ consists of matrices or tuples of an inner type S, then take $\preceq_T$ to be lexicographical order $\preceq_{Slex}$ over an order $\preceq_S$ for the inner type;*
5. *if $\mathrm{Val}(T)$ consists of multisets of type S, take $\preceq_T$ to be the $\preceq_M$ above.*

Note again that using Remark 1, this definition gives an ordering for *all* Essence types. An ordering for all types in other modelling languages can be defined in a similar way, by defining compound types in terms of multisets, tuples and matrices, and defining a concrete total ordering on each atomic type.

Remark 2. We can therefore refine constraints of the form $X \preceq Y$ to: $X \leqslant Y$ for the appropriate atomic ordering $\leqslant$ if X and Y are atomic; $X \preceq_{lex} Y$ if $\mathrm{Val}(X)$ and $\mathrm{Val}(Y)$ are matrices or tuples, and to $[-\mathrm{freq}(X,i)|i \in X] \preceq_{lex} [-\mathrm{freq}(Y,i)|i \in X]$, using an ordering of X, if they are (multi)sets, where $\mathrm{freq}(X,i)$ gives the number of occurrence of i in X. As the base cases in Definition 4 are ordering $\leqslant$ of atomic types, these $\preceq_{lex}$ will eventually be rewritten to $\leqslant_{lex}$.

4 Implementation in Conjure

We outline the implementation of the symmetry breaking method described in this paper within Conjure. The complete implementation can be found in the Conjure repository at `https://github.com/conjure-cp/conjure`.

Permutations We introduce a new Essence type `permutation`. We allow permutations of integers, enumerated types or unnamed types. Permutation is available as a domain constructor, using keywords `permutation of`, and takes either an integer, enumerated or unnamed type domain as argument. Permutation values and expressions are written in cycle notations. Each permutation has an attribute `NrMovedPoints`, which gives the number of points that are not fixed by the permutation.

Permutations can be naturally represented as bijective total functions, which in turn can be represented as 1-dimensional matrices, where the element at a certain index gives the image of the index. Each permutation is stored with its inverse. This is because it turns out we almost always need to use the inverse of a permutation during symmetry breaking, and storing both the permutation and its inverse was the most efficient option in practice. The operator `permInverse` gets the inverse of a permutation. It uses the fact that the inverse of the inverse of a permutation is the original permutation, so there is no need to calculate any further permutation applications when calling `permInverse` twice.

The operator `image` takes a permutation g on a type T and a value x of type T such that `image`(g, x) gives the image of x under g. The more general operator `transform`(g, X) represents the image of the induced action of g on the values of its second argument. If g is a permutation on a type T, and the type of X contains no reference to T, then `transform`(g, X) is rewritten to X.

Unnamed Types Members of an unnamed type can only be used as operands of an equality expression with other values of the same unnamed type. Unnamed types domains, similar to enumerated types, are eventually converted to `int(1..s)` where s is the size of the unnamed type. During refinements, unnamed types are converted to *tagged integers*, which behaves like integers but remembers the name (`IntTag`) of the unnamed type it comes from. The tags are important for correct permutation applications, and we are careful to only add valid symmetry breaking constraints on each type of tagged integer. Permutations on an unnamed type T are refined into permutations on integers tagged with T, and `image` and `transform` will only change integer values or variables with the appropriate tag. The refinement rules of Conjure ensure that when tagged integers are refined, the tag is preserved. For example a set of integers tagged with T is refined into a matrix indexed by integers tagged with T of Booleans. As seen in Proposition 1, each declaration of an unnamed type T will be removed, and all other domains constructed using T are replaced with tagged (with T) integers.

Symmetry Breaking Constraints As discussed in Section 3.3, we break the symmetries induced by unnamed types using lex-leader constraints. Our implementation allows sound but incomplete symmetry breaking by replacing the group in LL with a subset, which is determined by run-time flags. For a model with unnamed types $T_1, T_2, \ldots, T_k$ and decision variables $V_1, V_2, \ldots, V_n$, the first set of flags defines the subsets of the unnamed symmetries $\mathrm{Sym}(T_i)$ to be used in the lex-leader: for each $i \in \{1, 2, \ldots, k\}$, we take $S_i := \{(j_{T_i}, (j+1)_{T_i}) \mid 1 \leqslant j < |T_i|\}$ if with the `Consecutive` flag; $S_i := \{(t, u) \mid t, u \in T_i\}$ with the `AllPairs` flag; and $S_i := \mathrm{Sym}(T_i)$ with the `AllPermutations` flag. The second set of flags determines whether to take the product or the union of these S_i's: the constraint used is $\bigwedge_{1 \leqslant i \leqslant k} LL(S_i, (V_1, V_2, \ldots, V_n))$ if with the `Independently` flag; and is $LL(S_1 \times S_2 \times \ldots \times S_k, (V_1, V_2, \ldots, V_n))$ if with the `Altogether` flag. Each $LL(S, X)$ is expressed as the conjunction, over all possible permutations $g \in S$, of expressions of form X `.<=` $\texttt{transform}(g, X)$. Here `.<=` represents the total order $\preceq_T$ from Definition 4, where T can be any suitable type.

Permutation Application Starting from X `.<=` $\texttt{transform}(g, X)$ from above, if g is a list of permutations $[g_1, g_2, \ldots, g_r]$ representing an element of a direct product, we rewrite expressions of the form X `<=` $\texttt{transform}([g_1, g_2, \ldots, g_r], X)$ to the conjunction of X `.<=` x_i and $x_i = \texttt{transform}(g_i, x_{i-1})$ for $1 \leqslant i \leqslant k$, where x_0 is X and the x_i's are new variables. The refinement rules for $\texttt{transform}(g, x)$, when g is a permutation, follow from Definition 2. Conjure's general design, which applies rewrite rules until all high-level types and operators are removed, can easily handle this new set of rules. E.g. for a matrix X, we rewrite each entry $\texttt{transform}(g, X)[i]$ to $\texttt{transform}(g, X[\texttt{transform}(\texttt{permInverse}(g), i)])$. These internal `transform` and `permInverse` are further refined, until all permutations have been removed.

Refining Ordering Constraints Each constraint of the form X `.<=` Y is refined to $\texttt{symOrder}(X)$ `.<=` $\texttt{symOrder}(Y)$. Here $\texttt{symOrder}(X)$ signifies that we are to rewrite the expression using Remark 2. So X `.<=` Y will eventually be written to expressions of the form X' `<=lex` Y' or X' `<=` Y' for some X' and Y', where `<=` is the order of atomic types and `<=lex` the lexicographic ordering over `<=`. The lexicographic constraints will typically contain every variable, but can often

Table 1. How unnamed types occur in some problems

Problem	Type
Lam's Problem [28]	`matrix indexed by` [T,T] `of ?`
Set-theoretic Yang-Baxter [3]	`matrix indexed by` [T,T] `of` T
Balanced Incomplete Block Design [22]	`matrix indexed by` [T_1, T_2] `of ?`
Social Golfers [14]	`matrix indexed by` [T_1, T_2] `of` T_3
Covering Array [26]	`matrix indexed by` [T_1, T_2] `of` T_3
Template Design [27]	`matrix indexed by` [T_1] `of ?` `matrix indexed by` [T_1, T_2] `of ?`
Rack Configuration [15]	`function` T $\rightarrow$ `?`
Semigroups [8]	`function` (T, T) $\rightarrow$ `T`
Vellino's Problem [1]	`function:` T_1 $\rightarrow$ T_2 `function:` T_1 $\rightarrow$ `mset of` T_3
Sports Tournament Scheduling [29]	`relation of` (T_1 `*` T_2 `* set of` T_3)

be simplified, e.g. $[a, b, c, d] \leqslant_{lex} [a, d, b, d]$ can be simplified to $[b, c] \leqslant_{lex} [d, b]$. Rather than perform these simplifications while initially generating and refining the lexicographic ordering constraints, a set of general rules for simplifying lexicographic ordering constraints [30] is run after refinement is finished. Finally the model undergo further refinements until only types in Essence Prime remain.

5 Case Studies and Discussion

We give a few case studies to demonstrate our implementation of the method. Future work will include more in-depth experimentation. We consider the problems from Section 3.1, and three further problems with matrix types as decision variables, but with varying number of unnamed types occurring in various positions, as well as some examples where the decision variables are of different types. The types of the decision variables of these problems are summarised in Table 1, where the T and T_i's are all distinct unnamed types, and '?' denotes other types that are not constructed from any unnamed types. The models in Essence, and the automatically generated Essence Prime models, for all combinations of flags, can be found at `https://github.com/stacs-cp/CPAIOR2025-Symmetry`. The resulting models were manually inspected for correctness. In particular, the number of solutions for small instances of the set-theoretic Yang-Baxter equation and the semigroup problem are consistent with those in the literature [19,20].

The different methods of symmetry breaking provide an easy way of choosing between different trade-offs. `Altogether-AllPermutations` will break all symmetries, producing an exact list of symmetry-broken solutions, at the cost of a very large number of constraints. The fact that we need many constraints is not surprising, as the symmetries of unnamed types include several cases which have been proved theoretically difficult. Consider the solutions to a problem with a decision variable of type `matrix indexed by [T,T] of bool`, with no constraints. These solutions can be viewed as directed graphs, so completely breaking the unnamed type symmetries is equivalent to finding the canonical image of these graphs. Similarly, a `set of set (size 2) of T` can be viewed as the edges of an undirected graph. Checking if two solutions of these problems are equivalent is therefore as hard as the graph isomorphism problem. Further, a variable of type `matrix indexed by [T1,T2] of bool` has row and column symmetries, and efficient generation of complete symmetry breaking constraints is also at least as hard as the graph isomorphism problem [4].

If we consider a matrix indexed by two unnamed types, e.g. in the balanced incomplete block design problem, then `Independently-Consecutive` produces "double-lex", one of the most widely used symmetry breaking methods [9]. The constraints generated by `Independently-AllPairs` have also been used in practice [11], and can lead to faster solving. Deciding exactly what level of symmetry breaking is best for a particular class of problem is future work.

6 Conclusion and Future Work

In this paper, we show how the symmetries of indistinguishable objects can be broken completely together with an implementation in Essence. We do so by introducing the new `permutation` type, as well as a total ordering for all possible types in Essence, which allows us to express and automatically generate symmetry breaking constraints for unnamed types. We have also seen how we can soundly but incompletely break unnamed type symmetries by controlling the permutations used in the lex-leader constraints. Our abstract treatments of types and unnamed type symmetries make our method generalisable to any other solving paradigms with indistinguishable objects. This paper also serves as a theoretical background for further work in this area.

Much further work awaits. The symmetry breaking method here can be prohibitively expensive in some cases. We therefore will investigate how some relaxations of the ordering constraints can be used to give faster symmetry breaking method. Furthermore, the static ordering described in Section 3.3 can also be a source of inefficiency. This is because, when rewriting multisets to their occurrence representations in Remark 2, the elements in $[-\mathrm{freq}(X,i)|i \in X]$ must be sorted according to the total ordering of the inner types. This can be particularly difficult when we have deeply nested types. In general, it is not possible to produce a single global ordering which can be refined to a simple and efficient set of constraints in all possible representations. We shall therefore investigate the use of representation-specific total orderings.

Note that the symmetry breaking techniques in this paper will work even when the symmetry groups on unnamed types are not necessarily symmetric groups. We shall explore how the new permutation type can be used by an expert user for more control on symmetry breaking, in allowing arbitrary permutation groups, and see how we can automatically handle the symmetries for commonly occurring permutation groups, such as the chessboard symmetry. Generally, we intend to perform a more extensive analysis of the symmetry breaking constraints produced from our methods.

Acknowledgments. This work was supported by the Engineering and Physical Sciences Research Council EP/Y000609/1.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akgün, Ö.: CSPLib problem 116: Vellino's problem. `http://www.csplib.org/Problems/prob116`
2. Akgün, Ö., Frisch, A.M., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Conjure: Automatic generation of constraint models from problem specifications. Artificial Intelligence **310** (2022)
3. Akgün, Ö., Mereb, M., Vendramin, L.: Enumeration of set-theoretic solutions to the Yang–Baxter equation. Mathematics of Computation **91**(335), 1469–1481 (2022)
4. Anders, M., Brenner, S., Rattan, G.: The complexity of symmetry breaking beyond lex-leader. In: Principles and Practice of Constraint Programming (2024)
5. Bakewell, A., Frisch, A.M., Miguel, I.: Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In: International Workshop on Modelling and Reformulating CSPs (2003)
6. Baxter, N., Chu, G., Stuckey, P.J.: Symmetry declarations for minizinc. In: Proceedings of the Australasian Computer Science Week Multiconference. pp. 1–10 (2016)
7. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning. vol. 96, pp. 148–159 (1996)
8. Distler, A., Jefferson, C., Kelsey, T., Kotthoff, L.: The semigroups of order 10. In: Milano, M. (ed.) Principles and Practice of Constraint Programming. pp. 883–899. Springer Berlin Heidelberg (2012)
9. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Principles and Practice of Constraint Programming. pp. 462–477. Springer (2002)
10. Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proceedings of the Ninth National Conference on Artificial Intelligence. pp. 227–233 (1991)
11. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: Principles and Practice of Constraint Programming. p. 93–108. Springer-Verlag (2002)
12. Frisch, A.M., Miguel, I., Kiziltan, Z., Hnich, B., Walsh, T.: Multiset ordering constraints. In: International Joint Conferences on Artificial Intelligence. vol. 3, pp. 221–226 (2003)
13. Gent, I.P., Petrie, K.E., Puget, J.F.: Chapter 10: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 329 – 376. Elsevier (2006). `https://doi.org/https://doi.org/10.1016/S1574-6526(06)80014-3`
14. Harvey, W.: CSPLib problem 010: Social golfers problem. `http://www.csplib.org/Problems/prob010`
15. Kiziltan, Z., Hnich, B.: CSPLib problem 031: Rack configuration problem. `http://www.csplib.org/Problems/prob031`

16. Kiziltan, Z., Walsh, T.: Constraint programming with multisets. In: International Workshop on Symmetry in Constraint Satisfaction Problems. pp. 9–20 (2002)
17. Margot, F.: Symmetry in integer linear programming. 50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art pp. 647–686 (2009)
18. Mears, C., Niven, T., Jackson, M., Wallace, M.: Proving symmetries by model transformation. In: Principles and Practice of Constraint Programming. pp. 591–605. Springer (2011)
19. OEIS Foundation Inc.: The number of non-degenerate involutive set-theoretic solutions of the Yang-Baxter equation of order n up to isomorphism, Entry A290887 in The On-Line Encyclopedia of Integer Sequences (2025), `http://oeis.org/A290887`
20. OEIS Foundation Inc.: Number of nonisomorphic semigroups of order n, Entry A027851 in The On-Line Encyclopedia of Integer Sequences (2025), `http://oeis.org/A027851`
21. Poole, D.: First-order probabilistic inference. In: International Joint Conference on Artificial Intelligence. p. 985–991. Morgan Kaufmann Publishers Inc. (2003)
22. Prestwich, S.: CSPLib problem 028: Balanced incomplete block designs. `http://www.csplib.org/Problems/prob028`
23. Roy, P., Pachet, F.: Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In: Workshop on Non Binary Constraints (1998)
24. Sabharwal, A.: SymChaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints **14**, 478–505 (2009)
25. Sakallah, K.A.: Chapter 13: Symmetry and satisfiability. In: Biere, A., Heule, M., van Maaren, H. (eds.) Handbook of Satisfiability: Second Edition. Frontiers in Artificial Intelligence and Applications, IOS Press (2021)
26. Selensky, E.: CSPLib problem 045: The covering array problem. `http://www.csplib.org/Problems/prob045`
27. Smith, B.: CSPLib problem 002: Template design. `http://www.csplib.org/Problems/prob002`
28. Walsh, T.: CSPLib problem 025: Lam's problem. `http://www.csplib.org/Problems/prob025`
29. Walsh, T.: CSPLib problem 026: Sports tournament scheduling. `http://www.csplib.org/Problems/prob026`
30. Öhrman, H.: Breaking Symmetries in Matrix Models. MSc thesis, Department of Information Technology, Uppsala University (2005)