The Concept and Provenance of Unnamed, Indistinguishable Types

Alan M. Frisch[1]
Ian Miguel[2]
[1] Department of Computer Science, University of York, UK.
[2] School of Computer Science, University of St Andrews, UK.
frisch@cs.york.ac.uk, ianm@dcs.st-and.ac.uk

12th September, 2006
(very minor revisions: 19 September, 2006)

1 Introduction

Unnamed, indistinguishable types (UITs) are a useful feature in a constraint specification language since they enable the specification of problems at a higher level of abstraction. There is, however, some debate as to the concept of UITs and their provenance. This note is intended to clarify this matter. Specifically, it describes UITs, shows that they originated in our work [1] on ESSENCE, and shows that a related mechanism in ESRA is not as powerful.

2 Unnamed, Indistinguishable Types and Their Significance

To introduce and illustrate UITs we largely recount and quote from what is, to the best of our knowledge and belief, the first paper in which the concept appeared in the constraints literature, by Bakewell, Frisch and Miguel [1]. The Bakewell *et al* paper is about *refinement*, the process of producing a constraint model from an abstract constraint specification. Throughout this note, the *Balanced Incomplete Block Design* (BIBD) problem (problem 28 at `www.csplib.org`) is used as an example. The following is an English definition of the BIBD problem:

> Given positive integers, v, b, r, k and λ, arrange v distinct objects in b blocks such that each block contains k distinct objects, each object occurs in exactly r different blocks and every two distinct objects occur together in exactly λ blocks.

With this problem in mind, we now turn our attention to our ESSENCE specification language.[1] A problem specification is a triple consisting of a list of *declarations*, a set of *decision variables* and a set of *constraints*. We write this triple as:

[1] The language described in [1] is an early version of the ESSENCE [4, 5] language, which was unnamed at that time. This note uses the early version of ESSENCE as presented in [1].

`given`	$v, b, r, k, \lambda : \texttt{nat}$
`letting`	B be type-of-size b, V be type-of-size v
`find`	$BIBD : \wp(B \times V)$
`such that`	$\forall b{:}B.\ \lvert BIBD(b, \_)\rvert = k$
	$\forall v{:}V.\ \lvert BIBD(\_, v)\rvert = r$
	$\forall \{v, v'\}{:}\wp(V).\ \lvert BIBD(\_, v) \cap BIBD(\_, v')\rvert = \lambda$

Figure 1: An ESSENCE specification of the BIBD problem.

declarations `find` *decision variables* `such that` *constraints*

The list of declarations is used to define *parameters* and *constants*. The parameters to a problem specify the instance of the problem class. The value of a parameter is provided by the user in specifying the problem instance to be solved. The BIBD problem has parameters v, b, r, k, and λ. The decision variables denote the combinatorial objects that are to be found by the solver. The goal in the BIBD problem is to find a relation, which we shall call $BIBD$, between the blocks and the objects.

ESSENCE is strongly typed and every expression has a type. The five parameters in the BIBD problem are of type *natural*. The type of relation variable $BIBD$ is $B \times V$, hence its domain is $\wp(B \times V)$, where B and V are primitive types representing the blocks and objects. B and V are constant types and therefore must be declared in the list of declarations. Putting these pieces together, we obtain a specification of the BIBD as shown in Figure 1.

We see here that *declarations* is a list of `given` statement and `letting` statements; `given` statements declare the parameters and `letting` statements declare the constants (which may themselves be parameterised). B and V are declared to be primitive types. As in many constraint specification languages, ESSENCE allows a type (or domain) to be specified by enumerating the members. Unlike other languages, a type can also be specified by giving its size and not enumerating its members.

Constraints in ESSENCE are built from the declared parameters, constants and decision variables using operators commonly found in mathematics and in other constraint specification languages. ESSENCE also includes universally quantified variables ranging over any specified finite type. Each subexpression in a constraint is typed. Here $\{v, v'\} : \wp(V)$ is a two-element set containing objects of type V, and $BIBD(b, \_)$ is the projection of $BIBD$ onto b—that is, $\{v{:}V \mid BIBD(b, v)\}$.

We quote verbatim the following passages from [1]:

> Refinement introduces symmetries when it creates a model that distinguishes between objects that are not distinguished in the original problem formulation. For example, the original statement of the BIBD problem refers to a set B of blocks and a set V of objects, but does not distinguish among the elements of each set; no particular block or object is named. In the 01 matrix model given in Section 3.1, the blocks and objects serve as indices of the two dimensional matrix $BIBD_{M01}$. Indices of a matrix are distinguished; one index refers to the first row/column of a matrix, another refers to the second row/column, and so forth. Hence $BIBD_{M01}$ has row and column symmetry: its rows can be interchanged and its columns can be interchanged [2].[2]

[2]More precisely, an assignment of values to the variables of the matrix is a solution to the problem if and only if it is a solution after swapping the values assigned to any two rows/columns.

> We now show how our refinement rules can be enhanced to to recognise the symmetries that they introduce and to incorporate appropriate symmetry-breaking techniques into the models they produce...
>
> ⋮
>
> We shall not elaborate further on the mechanisms for dealing with symmetry, but instead close by noting that refining a specification that includes indistinguishable objects inevitably introduces symmetry. In its basic conception, an instance of the finite domain constraint satisfaction problem consists of a set of *named* variables, each of which is associated with a finite domain of *named* values. Hence, refining an abstract specification with indistinguishable objects must name, and thus distinguish, the objects, thereby introducing symmetry.
>
> ⋮
>
> Effective refinement also demands that the specification language must provide sufficient abstraction so that a specification need not make unnecessary distinctions, such as distinguishing between objects that should be indistinguishable. The consequence of using such a language is that fewer symmetries are contained in the initial specification, but more are introduced by refinement, which we believe can be identified and broken automatically. This point is clearly exemplified in our treatment of the BIBD problem. The initial specification does not name the elements of B or V; names are introduced by the refinement process, which simultaneously recognises that this introduces symmetry into the model. We know of no constraint specification language that supports sets of unnamed objects. Without this facility, the job of recognising the interchangeability of the arbitrarily-named objects is the responsibility of the user.

3 Unnamed, Indistinguishable Types in ESRA?

There is some debate as to whether the UITs introduced in [1] were, in fact, predated by a related concept in the constraint specification language ESRA. However, the ESRA paper [2] published in the 2nd International Workshop on Modelling and Reformulating CSPs (the same venue as [1]) makes no mention whatsoever of either unnamed or indistinguishable objects. Symmetry is mentioned only once in discussing the motivation for relational modelling:

> Relational models are more amenable to constraint analysis. Detected properties as well as properties consciously introduced during compilation into lower-level programs, such as symmetry or bijectiveness, can then be taken into account during compilation ...

The following year, a revised version of this paper appeared [3] containing two paragraphs that do address this topic, without citing any previous work. The paper provides an ESRA specification of the BIBD problem, shown here in Figure 2, and then discusses it:

$$
\begin{array}{l}
\texttt{dom } \mathit{Varieties}, \mathit{Blocks} \\
\texttt{dom } r, k, \lambda : N \\
\texttt{var } BIBD : \mathit{Varieties}^{\,r} \times^{k} \mathit{Blocks} \\
\texttt{solve} \\
\quad \forall (v_1 < v_2 : \mathit{Varieties})\ \mathrm{count}(\lambda)(j : \mathit{Blocks} \text{ — } BIBD(v_1, j) \wedge BIBD(v_2, j))
\end{array}
$$

Figure 2: An ESRA specification of the BIBD problem, taken from [3].

> Note the different style of modelling sets of unnamed objects, via the separation of models from the instance data, compared to Figure 2. There we introduce two sets without initialising them at the model level, while here we introduce three uninitialised constants that are then used to arbitrarily initialise three domains of desired cardinalities. Both models can be reformulated in the other style. The benefit of such sets of unnamed objects is that their elements are indistinguishable, so that lower-level representations of relational decision variables whose domains involve such sets are known to introduce symmetries.

Later, in speaking of the compilation of the ESRA specification of the BIBD problem:

> Hence, symmetry-breaking code [10, 32] would have to be inserted, as indicated in Figure 4. Since such choices are postponed to the compilation phase in ESRA (see Section 2.4), any symmetries consciously introduced can be handled (automatically) in that process.

We now compare the ESRA concept of "sets of unnamed objects" (SUNOs), first discussed in 2004 [3], with the UITs introduced in 2003 [1]. The SUNOs of ESRA are not comparable to UITs of Essence for several reasons:

1. A UIT is a type, and its values are distinct from those in every other type. In ESRA they are not types; ESRA has only two types (called "primitive domains"): "finite extensionally-given set of new names or integers". Only the values of input parameters can be "unnamed" in the sense that the values are not known until they are input. But the input values must be either new names or integers.

2. The values of a UIT have no properties other than being equal or unequal to each other. Consequently the syntax of Essence severely restricts the way that unnamed types can be used. In ESRA, elements of a SUNO have properties: they are either integers or new names, which have an ordering over them. Hence, the solve statement of Figure 2 tests $v_1 < v_2$, where v_1 and v_2 are elements of the SUNO $Varieties$. The syntax of Essence does not permit such an operation for UITs.

3. UITs are useful for refinement. The syntax of essence ensures that the values of a UIT are in fact indistinguishable. As discussed above, this guarantees that in any refinement of an Essence specification the objects representing the values of the UIT are interchangeable. That is, the refinement is guaranteed to have a certain set of symmetries — no symmetry detection is needed to determine this.

   In contrast, since there are no restrictions on how the elements of a SUNO are used in an ESRA specification, a model resulting from refinement may or may

not have a symmetry. Determining this requires analysis, either of the original ESRA specification or of the resulting model. In the solve statement of the ESRA specification Figure 2 the constraint is symmetric in that interchanging v_1 and v_2 results in a logically equivalent constraint. However, this need not be the case, as illustrated by the constraint

$$\forall(v_1 < v_2 : Varieties)\ v_1 \in S \rightarrow v_2 \in S \tag{1}$$

where S is some expression denoting a set. In a model of a specification containing (1) the objects representing the elements of $Varieties$ are not interchangeable. Determining if the elements of a SUNO are interchangeable requires complex analysis (and is probably undecidable). In contrast, the elements of UITs are always interchangeable; no analysis is necessary.

In effect, ESRA is merely providing parameters whose values are not known until they are instantiated with their input values; it is *not* providing known but unnamed, indistinguishable values. In ESRA the values of a SUNO may be indistinguished or they may be distinguished (as shown above), but in Essence, the values of a UIT are indistinguishable – they can *never* be distinguished.

References

[1] A. Bakewell, A.M. Frisch, I. Miguel. Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. *Proceedings of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pp. 2–17, 2003. www.cs.york.ac.uk/~frisch/Reformulation/03/

[2] P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a Relational Language for Modelling Combinatorial Problems. *Proc. of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems: Towards Systematisation and Automation*, pp 63-77, 2003. www.cs.york.ac.uk/~frisch/Reformulation/03/

[3] P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In: M. Bruynooghe (editor), *LOPSTR'03: Revised Selected Papers*, pp. 214-232. Lecture Notes in Computer Science, volume 3018. Springer-Verlag, 2004. www.springerlink.com/index/R1CK0MB0KN5W0MRG

[4] A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Essence of Essence: A Constraint Language for Specifying Combinatorial Problems. *Proc. of the Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73-88, October 2005. http://www.cs.york.ac.uk/aig/constraints/AutoModel/

[5] A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, I. Miguel. The Design of Essence: A Constraint Language for Specifying Combinatorial Problems. To appear in *Proc. of the 20th International Joint Conference on Artificial Intelligence* (IJCAI), 2006. http://www.cs.york.ac.uk/aig/constraints/AutoModel/