Compiling High-Level Type Constructors in Constraint Programming

Pierre Flener, Brahim Hnich, and Zeynep Kızıltan

Computer Science Division, Department of Information Science
Uppsala University, Box 513, S – 751 20 Uppsala, Sweden
{Pierre.Flener, Brahim.Hnich, Zeynep.Kiziltan}@dis.uu.se

Abstract. We propose high-level type constructors for constraint programming languages, so that constraint satisfaction problems can be modelled in very expressive ways. We design a practical set constraint language, called ESRA, by incorporating these ideas on top of OPL. A set of rewrite rules achieves compilation from ESRA into OPL, yielding programs that are often very similar to those that a human OPL modeller would (have to) write anyway, so that there is no loss in solving efficiency.

1 Introduction

Optimisation problems — where appropriate values for the problem variables must be found within their domains, subject to some constraints, such that some cost function on these variables takes an optimal value — are ubiquitous in industry. Examples are production planning subject to demand and resource availability so that profit is maximised, air traffic control subject to safety protocols so that flight times are minimised, transportation scheduling subject to initial and final location of the goods and the transportation vehicles so that delivery time and fuel expenses are minimised, etc. A particular case are *decision problems*, where there is no cost function that must take an optimal value. They are collectively known as *constraint satisfaction problems* (CSPs). Many of these problems can be declaratively expressed as constraint programs and then be solved using constraint solvers.

However, effective *constraint programming* (or: *modelling*) is very difficult, even for application domain experts, and hence time-consuming. Moreover, many of these problems are ill-behaved, in the sense that it can be shown that solving them requires an amount of time that is worse than polynomial in the size of the input data, hence making solving times prohibitively long.

To address the *programming-time problem*, ever more expressive and declarative constraint programming languages are being designed, providing traditional algebraic notations (such as sums and products over indexed expressions) and useful datatypes (such as sets, arrays, and enumerations) to enable a more natural expression of the constraints, freeing the programmer thus more and more from traditional (and often low-level) computing obligations, such as the writing of iterative/recursive code or the encoding of concepts as numbers.

To address the *solving-time problem*, the default behaviour of the solver can be modified and implied constraints can be posted so as to reduce the search space. Such optional (but often necessary) practice is however a concession that fully declarative constraint programming is still far away, and the question whether procedural search statements can be automatically added upon analysis of the constraints remains essentially open (but see [7, 10]).

Concerns about the solving time also require trade-offs about expressiveness: the programming language must after all be executable (though need not be computationally complete) and its programs should ideally execute quickly (and finitely). For instance, set constraint languages may well allow the formulation of constraints over sets (such as CLPS [1], CONJUNTO [8], NP-SPEC [3], OZ [11], and {LOG} [4]), hence providing enormous expressiveness, but if they cannot be compiled into acceptably fast code, then the advantage of decreased programming time is neutralised by the disadvantage of increased solving time.

Starting from the very expressive (and fast) OPL (Optimisation Programming Language) [16], we here design an even more expressive (and equally fast) language, called ESRA, and show how it is compiled into OPL. Like OPL, the ESRA language is strongly typed, and a sugared version of what is essentially a first-order logic language. Unlike OPL, the ESRA language supports more advanced types, such as mappings, and allows variables of these types as well as of type set, making it a set constraint language.

This paper is organised as follows. In Section 2, we present a motivating example. We can then introduce, in Section 3, the syntax of our ESRA language, as well as explain, in Section 4, the semantics of ESRA by showing how it is compiled into OPL. Finally, in Section 5, we conclude, compare with related work, and discuss our directions of future work.

2 A Motivating Example

We now argue that it is possible to improve the expressiveness of even OPL. After giving a (published) OPL model for a motivating example, we identify expressiveness problems with OPL, propose a more expressive model in our language, called ESRA, and show that the OPL model into which it compiles is very similar to the one initially given. To make this paper self-contained, no prior knowledge of OPL is assumed here and we explain all its features that are used here.

In the *Warehouse Location* problem [16], a company considers opening warehouses on some candidate locations in order to supply its existing stores. Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply (C_1). Each store must be supplied by exactly one open warehouse (C_2). The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimised.

```
int MaintCost = ...;
int NbStores = ...;
enum Warehouses ...;
range Stores 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var int OpenWarehouses[Warehouses] in 0..1;
var Warehouses Supplier[Stores];
minimize
  sum(I in Stores) SupplyCost[I,Supplier[I]]
  + sum(J in Warehouses) MaintCost * OpenWarehouses[J]
subject to {
  forall(I in Stores)
    OpenWarehouses[Supplier[I]]=1;
  forall(J in Warehouses)
    (sum(I in Stores) (Supplier[I]=J)) <= Capacity[J];
};
```

Fig. 1. Published OPL model of the Warehouse Location problem

2.1 An OPL Model of the Warehouse Location Problem

This problem can be modelled in OPL as in Figure 1, which is (a renaming of Statement 12.1) published in [16]. Instance data `MaintCost` and `NbStores` are integers read in at run-time, and so are the enumeration `Warehouses` of candidate warehouse locations, the (1-dimensional) array `Capacity` with the integer capacities of the warehouses, and the (2-dimensional) array `SupplyCost` with the integer supply costs to the stores from the warehouses. The type `Stores` is an integer range denoting the existing stores. The set of warehouses to be opened is modelled as an array `OpenWarehouses[Warehouses]` of Boolean variables, such that `OpenWarehouses[W]` is 1 if warehouse `W` is open. Also, the desired mapping from `Stores` into `Warehouses` is modelled by an array `Supplier[Stores]` of variables ranging in `Warehouses`, such that `Supplier[S]` is `W` when warehouse `W` supplies store `S`; this representation choice captures the "exactly one" part of constraint C_2. The `minimize` statement expresses that the addition of the sum of the supply costs and the sum of the maintenance costs (for the actually opened warehouses) must be minimal. The first `forall` statement expresses the "open" part of constraint C_2, while the second `forall` statement captures C_1. A nested constraint, such as `(Supplier[I]=J)`, is seen as 1 if true, and 0 if false.

Let us analyse this OPL model. Since set variables are not available,[1] the modeller had to find another way of expressing that a subset of the warehouses is to be found. The classical ILP (integer linear programming) way of modelling a subset of a given set as an array of Boolean variables was used. Therefore, a mapping from the set of stores into the *entire* set of warehouses had to be sought,

[1] Like ILOG Solver, OPL only supports *ground* sets, over *any* type (not just integers). OPL sets are thus only available for instance data, but *not* for domain variables, and OPL set operations thus only serve the pre-processing of instance data.

```
 1:  int MaintCost = ...;
 2:  int NbStores = ...;
 3:  enum Warehouses ...;
 4:  range Stores 0..NbStores-1;
 5:  int Capacity[Warehouses] = ...;
 6:  int SupplyCost[Stores,Warehouses] = ...;
 7:  var {Warehouses} OpenWarehouses;
 8:  var Stores->OpenWarehouses Supplier;
 9:  minimize
10:    sum(I->J in Supplier) SupplyCost[I,J]
11:    + card(OpenWarehouses) * MaintCost
12:  subject to {
13:    forall(J in OpenWarehouses)
14:      count(I in Stores: I->J in Supplier) <= Capacity[J];
15:  };
```

Fig. 2. An ESRA model of the Warehouse Location problem

instead of a mapping from the set of stores into a *subset* of the set of warehouses. This rather low-level data modelling forced a new way of perceiving the problem, leading to a rather awkward modelling of its cost function and constraints. Indeed, in the cost function, the sum of the maintenance costs has to be over *all* the warehouses, with the Booleans of `OpenWarehouses` being reinterpreted as weights, instead over just the *open* warehouses. Also, the first `forall` constraint is entirely due to the inability of the data modelling to express that a mapping from a given set to a *subset* of another given set has to be sought. The second `forall` constraint is above reproach, however.

2.2 An ESRA Model of the Warehouse Location Problem

Our ESRA language allows CSP modelling at an even higher level of abstraction than OPL. Introducing (among others) set and mapping variables, constraints over sets and mappings, a `card` function returning the cardinality of a set, and a `count` operator counting the number of times a relation holds, we propose the ESRA model in Figure 2 as a more expressive formalisation of the Warehouse Location problem. (Line numbers were added for future reference.) The variable declarations elegantly express that `OpenWarehouses` is a subset of `Warehouses`, and that `Supplier` is a mapping from `Stores` into `OpenWarehouses`, and this *without* the modeller having had to worry about their internal representations. A more natural formulation of the cost function and constraint C_1 arises from this, as well as a complete capture of constraint C_2 by the data modelling.

The OPL model generated from that ESRA model is given in Figure 3. (Line numbers were added for future reference.) Note the similarity of this OPL model with the OPL model in Figure 1. The declaration parts are the same, and so are the optimisation parts (except that our generated OPL model exploits distributivity of multiplication over addition). In the constraint parts, the first `forall` constraints are identical, while the second `forall` constraints differ a bit because

```
a:  int MaintCost = ...;
b:  int NbStores = ...;
c:  enum Warehouses ...;
d:  range Stores 0..NbStores-1;
e:  int Capacity[Warehouses] = ...;
f:  int SupplyCost[Stores,Warehouses] = ...;
g:  var int OpenWarehouses[Warehouses] in 0..1;
h:  var Warehouses Supplier[Stores];
i:  minimize
j:    sum(I in Stores) SupplyCost[I,Supplier[I]]
k:    + (sum(J in Warehouses) OpenWarehouses[J]) * MaintCost
l:  subject to {
m:    forall(I in Stores)
n:      OpenWarehouses[Supplier[I]]=1;
o:    forall(J in Warehouses)
p:      OpenWarehouses[J]=1 =>
q:        (sum(I in Stores) (Supplier[I]=J)) <= Capacity[J];
r:  };
s:  display(I in Warehouses: OpenWarehouses[I]=1) <I>;
```

Fig. 3. Generated OPL model of the Warehouse Location problem

the original OPL model iterates over all warehouses whereas the proposed ESRA model iterates only over the open warehouses (in fact, we could have chosen the same iteration in the ESRA model, but we believe that it is more natural to be only interested in checking the capacity constraints on the open warehouses). Furthermore, the generated OPL model has a display part to pretty-print the result. The (time and space) execution behaviours are identical.

3 The Syntax of ESRA

We now explain the design decisions behind ESRA, introduce its syntax, and motivate the need for some useful high-level type constructors.

3.1 Design Decisions

Since OPL is arguably the most expressive constraint programming language available nowadays, we decided to minimise our efforts for compiling ESRA into some executable form by using OPL as target language. Also, since it is arguably not frequently possible to improve on the expressiveness of OPL, a natural choice was to make ESRA a conservative extension of OPL, so that entire passages of ESRA programs can be literally copied during compilation. Just like the designers of OPL, we do not really care whether our ESRA language is complete (in any sense) or not, our main driving force being rather the design of a language that is practical for modelling at least some classes of (real-life) CSPs.

ESRA is an extension of OPL because we introduce useful high-level type constructors and allow the set operators of OPL in set constraints.[2] ESRA is thus designed to be more expressive than even OPL, and we will show that this can be done without compromising on efficiency.

These design decisions allow us to benefit, as a side-effect, from the fact that the OPL syntax elegantly hides that OPL actually is a logic language. Indeed, typed quantifications are replaced by C-like type and variable declarations, conjunction is denoted by a semi-colon (the usual notation in imperative programming for sequential composition), etc. It is unfortunate that plain logic notation is considered repulsive by many programmers, so efforts indeed must be undertaken to give them a language with the look and feel of other languages.

3.2 Syntax

Ignoring search issues, an ESRA program consists of a declaration part, followed by an optional optimisation part, and a constraint part, as described next.

In the *declaration part*, the syntax of OPL is applied to declare user-defined types, as well as typed instance data and variables. Instance data can be initialised in the usual OPL ways, at compile-time or at run-time. The primitive types are the integers (`int`), enumerations (`enum`), and strings (`string`) of OPL. The type constructors are the ranges (`range`), records (`struct`), arrays (`array`), and sets ({}) (over *any* type) of OPL, as well as new ones (described in the next sub-section), namely a binary constructor (written `->` and used in an infix way) for mappings between sets, a unary constructor (`perm`) for permutations of sets, and a binary constructor (`seq`) for sequences of bounded length over sets. Contrary to OPL, there can even be declarations of variables of type set in ESRA. For lack of space, we refer to [6] for the BNF grammar of the declaration part. See lines `1` to `8` of Figure 2 for a sample declaration part.

In the *optimisation part* and *constraint part*, the syntax of OPL is again used, this time to express the cost function that has to be optimised, and to post constraints. The main primitive constraints, relations, and expressions are the usual ones for (integer) arithmetic, (Boolean) logic, and sets. Powerful aggregation operators such as summation (`sum`) and universal quantification (`forall`) are available, making more general iteration/recursion mechanisms largely unnecessary. Contrary to OPL, the set operators (`in`, `subset`, `union`, `inter`, `card`, etc) can also be used in ESRA constraints. Existential quantification (`exists`) and counting (`count`) are also new. We again refer to [6] for the BNF grammar of the optimisation and constraint parts. See lines `9` to `11`, and lines `12` to `15`, in Figure 2 for sample optimisation and constraint parts.

3.3 Useful High-Level Type Constructors

It is generally recognised that the highest-level data structures are:

[2] As our effort is not sponsored by ILOG, we do not infringe on copyrights by choosing a radically new name (ESRA) for our language, rather than something like OPL+.

- *sequences*: element containers where the union operation is associative, with element order and element repetition being relevant;
- *bags* (or *multisets*): sequences where the union operation is commutative, making element order irrelevant;
- *sets*: bags where the union operation is idempotent, making element repetition irrelevant.

Sequences, bags, and sets are of possibly unbounded cardinality. Their usage is recommended for the high-level modelling of problems. At lower levels of abstraction, these data structures can be represented in a variety of ways, using trees, bit vectors, pointers, etc. As we are (here) not interested in sequential access to sequence elements nor in sequences of unbounded cardinality, we abandon sequences in favour of (fixed-size) arrays, with direct access to elements. Similarly, we are (for the time being) not concerned with bags and infinite sets, and ignore them as modelling devices.

So, equipped with (finite) sets and arrays, what can a problem modeller do? More precisely, are there any useful recurring modelling idioms that can be captured in new ways? Following D.R. Smith [12], we claim that many problems are of either of the following four classes: [3]

- *SUBSET*: Find a subset of a given set. For example, finding a clique of a graph amounts to finding a subset of its vertex set.
- *MAPPING*: Find a mapping from a given set to another given set. For example, the colouring of the countries of a map, such that any two neighbour countries have different colours, fits this class.
- *PERMUTATION*: Find an array that represents a permutation of a given set. For example, scheduling jobs according to precedence constraints is a permutation problem.
- *SEQUENCE*: Find an array that represents a sequence, of bounded cardinality, of elements drawn from a given set. For example, a (variant of the) travelling salesperson problem can be modelled this way, with a set of cities being ordered into a route, such that every city is visited at least once.

Going beyond Smith's classification now, we recognise that many real-life problems are actually hybrid in nature, so that we also need to support any *combination* of the four classes above. For instance, the Warehouse Location problem is a hybrid of *SUBSET* and *MAPPING*, because a mapping has to be found from the given set of stores into a subset of the given set of warehouses.

The four classes above are thus actually not classes of stand-alone problems, but rather give rise to powerful high-level *type constructors*, of which several can be used in the same program. The syntax and (informal) meaning of their usage in variable declarations is as follows:

- `var {T} S`: Set `S` is a subset of set `T`. A superset of `T` must be *known* (i.e., either `T` is a set or `T` is a subset of a known set). The internal representation of sets is hidden from the modeller, without losing in power. For instance:

[3] Smith actually identifies seven classes, but we discarded one here, as it is not applicable to CSPs, and we twice merged two of his classes into one.

```
var {Vertices} Clique;
...
forall(A,B in Clique) <A,B> in Edges;
```

is the core of a model of the clique problem.

- `var V->W M`: Mapping `M` is from set `V` into set `W`. Supersets of `V` and `W` must be known. The internal representation of mappings is hidden from the modeller, access being restricted as in an abstract datatype. For instance:

```
var Countries->Colours M;
...
forall(A,B in Countries) <A,B> in Neighbours => M[A]<>M[B];
```

is the core of a model of the map colouring problem.

- `var perm(S) A`: Array `A` represents a permutation of set `S`. A superset of `S` must be known. For instance:

```
var perm(Jobs) Sched;
...
forall(I,J in 1..card(Sched)) <Sched[I],Sched[J]> in Prec => I<J;
```

is the core of a model of the job scheduling problem.

- `var seq(S,K) A`: Array `A` represents a sequence, of bounded cardinality `K`, of elements drawn from set `S`. A superset of `S` must be known. For instance:

```
int MaxCities = ...;
var seq(Cities,MaxCities) Route;
...
forall(City in Cities) City in Route;
```

is the core of a model of our travelling salesperson problem.

The *SUBSET* class can be usefully generalised to *nSUBSETS*, where the aim is to find a maximum of n subsets of the given set. For instance, the Warehouse Location problem can also be seen as finding, for each warehouse, the set of stores to which it delivers, i.e., finding `card(Warehouses)` subsets of `Stores`. (Note that these subsets must be disjoint and that their union must be `Stores`; however, this is not a partitioning problem, as some of the subsets may be empty, denoting the fact that some warehouses are not to be opened.)

4 The Semantics of ESRA

We now explain the semantics of the ESRA language, by exhibiting the architecture of a compiler (into OPL), and showing that the main modules of that architecture can be easily implemented by a set of ESRA-to-OPL rewrite rules. Finally, we give another example of the power of our approach, by modifying the Warehouse Location problem, re-modelling it straightforwardly in ESRA, but obtaining a less intelligible and longer OPL program through compilation.

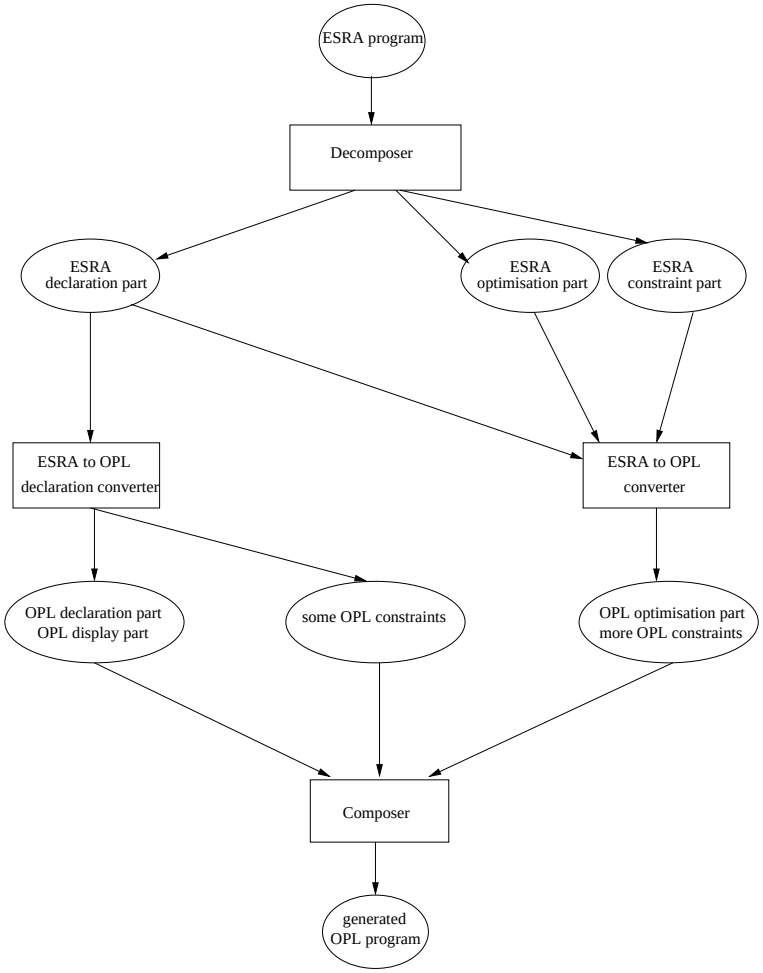

Fig. 4. Architecture of the ESRA compiler

4.1 Architecture of our ESRA Compiler

The architecture of our ESRA compiler is shown in Figure 4. First, the Decomposer separates an ESRA program into its declaration, optimisation, and constraint parts. Next, the ESRA-to-OPL Declaration Converter rewrites all ESRA declarations into OPL declarations, and possibly into some OPL constraints and OPL display statements (see Section 4.2 for details). Also, the ESRA-to-OPL Converter rewrites the ESRA optimisation and constraint parts into an OPL optimisation part and more OPL constraints, using the declaration part (again see Section 4.2 for details). Finally, the Composer assembles the generated OPL program by suitably concatenating the obtained OPL statements.

The Decomposer and Composer modules are trivial, and are not discussed here. The converter modules are explained next.

4.2 ESRA-to-OPL Rewrite Rules

We use conditional rewrite rules, here written as follows:

$$L \;\Rightarrow\; R \quad | \quad C$$

meaning that, if condition C holds, then expression L is rewritten into R.

As a running example, we show how each line of the ESRA model in Figure 2 is compiled into some line(s) of the OPL model in Figure 3. For reasons of space, we here only exhibit the rules that are needed to make this paper self-sufficient. We refer to [6] for the complete set of rules.

ESRA-to-OPL Declaration Converter. The declarations of ESRA that involve types not supported (in the same way) by OPL (namely sets, mappings, permutations, and sequences) are rewritten into OPL declarations, and possibly into some OPL constraints and display statements. All other declarations literally become OPL declarations. For instance, lines `a-f` are identical to lines `1` to `6`.

For set variable declarations, one of the rules is as follows:

```
var {T} S;
 ⇒  var int S[T] in 0..1; display(I in T: S[I]=1) <I>;
 |  T is a known set
```

A set `S` of known super-set `T` is thus represented, *in this case*, as an array of Boolean variables, indexed by `T`. This Boolean representation of sets is more memory-consuming than the set interval representation of CONJUNTO [8] and OZ [11], but both have been shown to create the same $O(2^n)$ search space [8]. Moreover, the set interval representation does not allow the definition of some (to us) desirable high-level primitives, such as universal quantification over elements of non-ground sets. This is why we have resorted to the Boolean representation, which is only naive in appearance. A display statement is also generated, in order to pretty-print `S`. For instance, line `7` is rewritten into lines `g` and `s` because `Warehouses` is declared in line `3` as an enumeration.

For mapping variable declarations, one of the rules is as follows:

```
var V->W M;
 ⇒  var T M[V]; forall(I in V) W[M[I]]=1;
 |  V is a known set, and W is a set variable of known super-set T
```

The mapping `M` is thus represented as an array of variables drawn from `T`, indexed by `V`. Furthermore, we post the constraint that every actually mapped element of `T` must be a member of `W`. For instance, line `8` is rewritten into lines `h`, `m`, `n`.

However, if a mapping is between set variables, then the rule is as follows:

```
var V->W M;
  ⇒  var int M[S,T] in 0..1;
     forall(I in S, J in T) M[I,J]=1 => V[I]=1 & W[J]=1;
     forall(I in S) V[I]=1 => (sum(J in T) (M[I,J]=1))=1;
   |  V and W are set variables of known super-sets S and T, respectively
```

The mapping `M` is thus represented as a two-dimensional array of Booleans, indexed by `S` and `T`. Furthermore, we post the constraint that every actual pair `<I,J>` of `M` forces `I` and `J` to be members of `V` and `W`, respectively. Finally, we post the constraint that every element of `V` must be mapped to exactly one element of `W`, because modelling the mapping as a Boolean matrix does not by itself enforce this. This rule will be used in Section 4.3.

For each declaration involving n sets, there are 2^n rewrite rules, depending on whether each set is itself a variable or not.

ESRA-**to-**OPL **Converter.** Expressions and constraints of ESRA that involve types not supported (in the same way) by OPL (namely sets, mappings, permutations, and sequences) are rewritten into OPL. The set operations of OPL (such as `union`, `inter`, `in`, `subset`) are thus now also allowed in constraints, rather than only in the pre-processing of instance data. Similarly for the expressions and constraints of ESRA that do not exist in OPL (such as `card`, `count`, `exists`).

For sums over mappings, one of the rules is as follows:

```
sum(I->J in M) F(I,J)
  ⇒  sum(I in V) F(I,M[I])
   |  V is a known set, W is a set variable of known super-set T,
      and M is a mapping from V into W
```

because, in this case, the mapping `M` is represented by an array of elements drawn from `T`, indexed by `V`. For instance, line `10` is rewritten into line `j`.

For the cardinality of a set, one of the rules is as follows:

```
card(S)
  ⇒  sum(I in T) S[I]
   |  S is a set variable of known super-set T
```

because, in this case, the set `S` is represented by a Boolean array, indexed by `T`. For instance, line `11` is rewritten into line `k`.

For membership in a mapping, one of the rules is as follows:

```
I->J in M
  ⇒  M[I]=J
   |  V is a known set, W is a set variable of known super-set T,
      and M is a mapping from V into W
```

because, in this case, the mapping `M` is represented by an array of elements drawn from `T`, indexed by `V`, and the set `W` is represented by a Boolean array, indexed by `T`. For instance, part of line `14` is rewritten into part of line `q`.

For count expressions over sets, one of the rules is as follows:

```
count(I in S: C)  ⇒  sum(I in S) C   |   S is a known set
```

because, in this case, the set `S` is known. For instance, part of line `14` is rewritten into part of line `q`.

For universal quantification over sets, one of the rules is as follows:

```
forall(I in S) P(I);
 ⇒  forall(I in T) S[I]=1 => P(I);
  |   S is a set variable of known super-set T
```

because, in this case, the set `S` is represented by a Boolean array, indexed by `T`. For instance, line `13` is rewritten into lines `o` and `p`.

4.3 Modifying the Warehouse Location Problem

Suppose we modify the Warehouse Location problem as follows (with modifications being highlighted in italics): A company considers opening warehouses on some candidate locations in order to supply its existing stores, *as well as possibly closing some of these stores, but such that a certain minimum number of stores remains open* (C_3). Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply (C_1). Each store *that is not closed* must be supplied by exactly one open warehouse (C_2'). The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open *and which stores not to close*, and which of these warehouses should supply the various stores *that are not closed*, such that the sum of the maintenance and supply costs is minimised.

In other words, we now look for a mapping of a *subset* of the stores into a subset of the warehouses. Figure 5 shows an ESRA model of this problem.

The OPL model generated from that ESRA model is shown in Figure 6. Note that the second rule for compiling mapping variables was used here. Comparing it with the original OPL models in Figures 1 and 3, we observe not only that new variable declarations and constraints were added, but also that some existing variable declarations and constraints had to be modified, with the overall code becoming quite complex. However, the higher-level of abstraction of ESRA allowed a very straightforward re-modelling, where only new variable declarations and constraints were added, compared to the original ESRA model in Figure 2, with the overall code still matching the informal problem specification very closely. This became possible because an ESRA modeller need not worry how mappings and subsets are internally represented.

5 Conclusion

Summary. Our contributions in this paper are (*i*) the proposal of high-level type constructors for constraint programming languages, so that CSPs can be modelled in more straightforward ways; (*ii*) the design of a practical set constraint language, called ESRA, incorporating these ideas; and (*iii*) the development of rewrite rules achieving a compilation from ESRA into OPL.

```
int MaintCost = ...;
int NbStores = ...;
enum Warehouses ...;
range Stores 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
int MinNbStores = ...;
var {Stores} RemainingStores;
var {Warehouses} OpenWarehouses;
var RemainingStores->OpenWarehouses Supplier;
minimize
  sum(I->J in Supplier) SupplyCost[I,J]
  + card(OpenWarehouses) * MaintCost
subject to {
  card(RemainingStores) > MinNbStores;
  forall(J in OpenWarehouses)
    count(I in Stores: I->J in Supplier) <= Capacity[J];
};
```

Fig. 5. An ESRA model of the modified Warehouse Location problem

We have shown that the resulting OPL programs are often very similar to OPL models written by human modellers. The key issue is of course that it is easier to write the ESRA model, because ESRA offers higher-level abstractions than OPL. Interestingly, this result comes at no cost to solving efficiency, precisely because these abstractions can be automatically mapped into OPL statements that a human OPL modeller would (have to) write anyway.

Because based on OPL, our ESRA language is computationally incomplete, but this does not disturb us, as we just aim at speeding up the modelling (and solving) of at least some classes of (real-life) CSPs. This philosophy is in line with the current trend on domain-specific tools, such as the PLANWARE [2] system for planning problems, or the primitives of OPL [16] for scheduling and resource allocation problems.

Related Work. Several set constraint languages exist (such as CLPS [1], CONJUNTO [8], NP-SPEC [3], OZ [11], and {LOG} [4]), but none of them seems as expressive (or fast) as our proposal, for lack of the rich data and constraint modelling mechanisms of OPL, and as none of them features all the additional type constructors we advocate here. The first three languages are limited to constraints on sets of initially known cardinality, and some of them do not support optimisation problems.

A language-independent computer-assisted constraint programming architecture was proposed [15], but it does not support set constraints.

Taking a completely different approach, D.R. Smith developed the program synthesisers KIDS [12, 13], DESIGNWARE [14], and PLANWARE [2], which semi-automatically compile high-level specifications written in REFINE into applicative programs. When applied to specifications of CSPs, these systems excel (at help-

```
int MaintCost = ...;
int NbStores = ...;
enum Warehouses ...;
range Stores 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
int MinNbStores = ...;
var int RemainingStores[Stores] in 0..1;
var int OpenWarehouses[Warehouses] in 0..1;
var int Supplier[Stores,Warehouses] in 0..1;
minimize
  sum(I in Stores, J in Warehouses) SupplyCost[I,J] * Supplier[I,J]
  + (sum(J in Warehouses) OpenWarehouses[J]) * MaintCost
subject to {
  forall(I in Stores, J in Warehouses)
    Supplier[I,J]=1 => RemainingStores[I]=1 & OpenWarehouses[J]=1;
  forall(I in Stores)
    RemainingStores[I]=1 => (sum(J in Warehouses) Supplier[I,J]) = 1;
  (sum(I in Stores) RemainingStores[I]) > MinNbStores;
  forall(J in Warehouses)
    OpenWarehouses[J]=1 => (sum(I in Stores) (Supplier[I,J]=1)) <= Capacity[J];
};
display(I in Stores: RemainingStores[I]=1) <I>;
display(I in Warehouses: OpenWarehouses[I]=1) <I>;
display(I in Stores, J in Warehouses: Supplier[I,J]=1) <I,J>;
```

Fig. 6. Generated OPL model of the modified Warehouse Location problem

ing) in the generation of high-speed problem-specific solvers, which have been deployed numerous times in industry, as they often outperform operations research or constraint programming solvers. Significant amounts of theorem proving and (computer-assisted) program optimisation are performed. These synthesisers support possibly infinite sequences, bags, and sets, and allow thus more expressive modelling than ESRA.

Our work is a result of adapting the fundamental ideas of KIDS and its successors to the generation of constraint programs (see [5] for an early report).

Future Work. In order to fulfill our design intention of making ESRA also more declarative than OPL, we are currently investigating the compile-time generation of also a (procedural) OPL search part by analysis of a (declarative) ESRA model that has no such search part. First-generation constraint solvers were black boxes and thus did not allow the formulation of model-specific labelling heuristics. The current second-generation solvers, such as the one for OPL, provide an elaborate notation for expressing such heuristics. As necessary as this may be, doing so remains more of an art than a science. We thus see no reason not to quietly prepare a third generation, where model-specific search parts in such a notation are actually synthesised. See [7, 10] for first results.

We also study the reformulation of ESRA models, by investigating the merits of alternative OPL representations of high-level data structures, by considering the integration of alternative models, and by examining the generation of implied constraints. See [9] for first results.

Acknowledgements

This research is partly funded under grant number 221-99-369 of TFR (the Swedish Research Council for Engineering Sciences). We are grateful to the referees for their useful comments.

References

1. F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *Techniques et Sciences Informatiques* 15(3):297–328, 1996.
2. L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold. PLANWARE: Domain-specific synthesis of high-performance schedulers. In *Proc. of ASE'98*, pp. 270–279. IEEE Computer Society Press, 1998.
3. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In: G. Gupta (ed), *Proc. of PADL'99*, pp. 16–30. LNCS 1551. Springer-Verlag, 1999.
4. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. On the representation and management of finite sets in CLP-languages. In: J. Jaffar (ed), *Proc. of JICSLP'98*, pp. 40–54. The MIT Press, 1998.
5. P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In *Proc. of ASE'98*, pp. 168–176. IEEE Computer Society Press, 1998.
6. P. Flener and B. Hnich. The Syntax and Semantics of ESRA. ASTRA Internal Report. Available via http://www.dis.uu.se/~pierref/astra/.
7. P. Flener, B. Hnich, and Z. Kızıltan. A meta-heuristic for subset problems. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*. LNCS, this volume. Springer-Verlag, 2001.
8. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.
9. B. Hnich and P. Flener. High-level reformulation of constraint programs. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.
10. Z. Kızıltan, P. Flener, and B. Hnich. A labelling heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.
11. T. Müller. Solving set partitioning problems with constraint programming. In *Proc. of PAPPACT'98*, pp. 313–332. The Practical Application Company, 1998.
12. D.R. Smith. The structure and design of global search algorithms. *Tech. Rep. KES.U.87.12*, Kestrel Institute, 1988.
13. D.R. Smith. KIDS: A semi-automatic program development system. *IEEE Trans. on Software Engineering* 16(9):1024–1043, 1990.
14. D.R. Smith. Toward a classification approach to design. In *Proc. of AMAST'96*, pp. 62–84. LNCS 1101. Springer-Verlag, 1996.
15. E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In: J. Little (ed), *Proc. of PACLP'99*, pp. 81–93. The Practical Application Company, 1999.
16. P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.