Enumerated Types and Type Extensions for MiniZinc

Peter J. Stuckey and Guido Tack

Department of Data Science and Artificial Intelligence, Monash University, Australia
{peter.stuckey,guido,.tack}@monash.edu

Abstract. Discrete optimisation problems often reason about finite sets of objects. While the underlying solvers will represent these objects as integer values, most modelling languages include enumerated types that allow the objects to be expressed as a set of names. Data attached to an object is made accessible through given arrays or functions from object to data. Enumerated types improve models by making them more self documenting, and by allowing type checking to point out modelling errors that may otherwise be hard to track down. But a *frequent modelling pattern* requires us to add new elements to a finite set of objects to represent extreme or default behaviour, or to combine sets of objects to reason about them jointly. Currently this requires us to map the extended object sets into integers, thus losing the benefits of using enumerated types. In this paper we introduce enumerated type extension, a restricted form of discriminated union types, to extend enumerated types without losing type safety, and default expressions to succinctly capture cases where we want to access data of extended types. The new language features allow for more concise and easily interpretable models that still support strong type checking and compilation to efficient solver-level models.

1 Introduction

Discrete optimisation models often reason about a set of given objects, and make use of data defined on those objects. In MiniZinc [9] (and other CP modelling languages) the core way of representing this information is as an *enumerated type* defining the objects, and arrays indexed by the enumerated type to store the data. Given that debugging constraint models can be quite difficult, particularly if the solver simply fails after a large amount of computation, an important role of enumerated types in modelling languages is to provide *type safety*. Many subtle errors can be avoided if we use strong type checking based on the enumerated types. Indeed MiniZinc and other languages such as Essence [4] provide strong type checking of enumerated types.

One of the greatest strengths of constraint programming modelling languages is the use of variable index lookups, i.e., looking up an array with a decision variable, supported in CP solvers by the `element` constraint. Variables in CP models are often declared specifically for this purpose. Accessing an array with an incorrect index is one of the most common programming mistakes, and replacing

integer index sets with enumerated types is a powerful technique that turns these mistakes into static compiler errors. This means that in order to index arrays with variables, we require variables that range over an enumerated type. Note that in the relational semantics [3] used by MiniZinc, the undefinedness from looking up an array at a non-existing index leads to falsity rather than a runtime abort, which may be difficult to detect if it does not occur in a root context (where the constraints have to hold). Hence, enumerated types and type checking are arguably even more important for constraint modelling languages.

However, real models are usually more complex, and quickly reach the limits of the current support for enumerated types. Although we define a set of objects to reason about, often modellers need to (a) add additional objects to the set to represent extreme or exceptional cases, and/or (b) reason about two sets of objects jointly. Currently we can resolve this problem by mapping the objects to integers and reasoning about index sets which are subsets of integers. But in doing so we lose the advantages of strong type checking.

In this paper we introduce mechanisms into MiniZinc that enable enumerated types to be defined by extending or joining other enumerated types, in a type-safe way. A review of the MiniZinc benchmark library reveals that many models can benefit from these new features.

2 Preliminaries

We give a brief introduction to MiniZinc in order to help parse the example code in the paper. A model consists of a set of variable declarations, constraints and predicate/function definitions, as well as an optional objective. Basic types (for our purposes) are integers, Booleans and enumerated types. MiniZinc also supports sets of these types. MiniZinc uses the notation $l..u$ to indicate the integer interval from l to u including the endpoints. We can define parameters and variables of these types, using a declaration [`var`] T: *varname* [`=` *value*] where T is a basic or set type or interval. Variable sets must be over integers or enumerated types. The optional *value* part can be used to initialise a parameter or variable. We can define multi-dimensional arrays in the form `array[`$indexset_1, indexset_2, .., indexset_n$`] of` [`var`] T`:` *arrayname* where each $indexset_i$ must be a range of either integers or an enumerated type.

One of the most important constructs in MiniZinc are array comprehensions written `[` e `|` *generator*(s) `]`, where e is the expression to be generated. Generator expressions can be i `in` S where i is a new iterator variable and S is a set, or i `in` a where a is an array. These cause i to take values in order from the set or array. Optionally they can have a `where` *cond* expression which limits the generation to iterator values that satisfy the condition *cond*. We concatenate one dimensional arrays together using the `++` operator.

Generator call expressions of the form f`(` *generators* `)(` e `)` are syntactic sugar for f`([` e `|` *generators* `])`. The most important functions used in generator call expressions are `forall` (conjoining the elements of the array), `exists` (disjoining the elements of the array) and `sum` (summing up the array elements).

Conditionals are of the form `if` *cond* `then` *thenexpr* `else` *elseexpr* `endif`. They evaluate as *thenexpr* if *cond* is true and *elseexpr* otherwise. Note that *cond* need not be a fixed Boolean expression, but may be decided by the solver [10].

Finally we occasionally use array slicing notation. In a two dimensional array a the expression $a[i, ..]$ returns the one dimensional array $[a[i, j] \mid j \in indexset_2]$ where $indexset_2$ is the second (declared) index set of array a.

3 Enumerated Types

An enumerated type is a simple type consisting of a named set of objects. Enumerated types are common to almost all programming languages as well as many modelling languages. They can be just syntactic sugar for integers, as in C; or they can be treated as distinct types by the type checker, as in Haskell, TypeScript, or MiniZinc, giving stronger checking of programs and models. Enumerated types are a special case of discriminated union types.

This section gives an overview of the existing enumerated type support in MiniZinc. Enumerated types are declared using the keyword `enum`. For example an enumerated type of colours might be

```
enum COLOUR = { Red, Orange, Yellow, Green, Blue, Violet };
```

which declares not only the type `COLOUR`, but six constant colour identifiers. These identifiers can then be used throughout the model. A model can also simply define the name of an enumerated type:

```
enum COLOUR;
```

which is then specified in a data file as

```
COLOUR = { Red, Orange, Yellow, Green, Blue, Violet };
```

Alternatively an *anonymous* enumerated type may be constructed using `anon_enum`. For example, imagine we are colouring a graph with n colours, we may use

```
COLOUR = anon_enum(n);
```

to specify the colours.

Definition 1. *In MiniZinc, an enumerated type is defined using the syntax*

$$\begin{aligned}
\langle \textit{enum-declaration} \rangle &\rightarrow \textbf{\texttt{enum}}\ \textbf{\textit{id}}\ [\texttt{=}\ \langle \textit{enum} \rangle\ ] \\
\langle \textit{enum} \rangle &\rightarrow \texttt{\{}\ \langle \textit{list-of-id} \rangle\ \texttt{\}} \\
\langle \textit{enum} \rangle &\rightarrow \textbf{\texttt{anon_enum}}\ \texttt{(}\ \langle \textit{expr} \rangle\ \texttt{)} \\
\langle \textit{list-of-id} \rangle &\rightarrow \langle \textit{list-of-id} \rangle\ \texttt{,}\ \textbf{\textit{id}} \\
\langle \textit{list-of-id} \rangle &\rightarrow \textbf{\textit{id}}
\end{aligned}$$

where ***id*** *is a MiniZinc identifier and expr is an (integer) expression. The identifiers defined in different enumerated types are required to be distinct.* □

Values of an enumerated type naturally represent a set of different, ordered (as given in the list) objects. Operators such as `=`, `!=`, `<`, `>=` and functions such as `min` and `max` have the natural definition on enumerated types. MiniZinc also supports the partial function `enum_succ` (the next element in the type) and `enum_prev` (the previous element in the type).

The most common use of enumerated types is as a set to iterate over in constraints. For example a simple knapsack problem can be defined by

```
enum PRODUCT;
array[PRODUCT] of int: price;
array[PRODUCT] of int: profit;
array[PRODUCT] of var bool: chosen;
constraint sum(i in PRODUCT)(price[i]*chosen[i]) <= budget;
solve maximize sum(i in PRODUCT)(profit[i]*chosen[i]);
```

One of the great strengths of CP modelling is the use of global constraints. While global constraints are defined on integers, we often want to apply them to enumerated types. In MiniZinc this is accomplished by treating enumerated types as subtypes of integers, and automatically coercing them to integers when required. For example in the model where we are ordering people in a line

```
enum PERSON;
enum ORDER = anon_enum(card(PERSON));
array[ORDER] of var PERSON: x;
constraint alldifferent(x);
```

the `alldifferent` constraint acts on `PERSON` which are automatically coerced into the integers $\{1, \ldots, n\}$ where n is the number of elements in `PERSON`. This also applies when we apply arithmetic operations, e.g. the successor function is similar in effect to $x + 1$, which has the effect of taking an enumerated type value x, coercing it to an integer and adding one, returning an integer. In order to map back from integers, MiniZinc supports the `to_enum` partial function which maps an integer back to an enumerated value (when possible), e.g. `x = to_enum(PERSON, y+1)` returns the successor of `PERSON` y. According to the relational semantics of MiniZinc, `to_enum` will become false in the enclosing Boolean context if the given integer is outside of the valid values of the enumerated type.

One of the reasons that enumerated types are critically important to CP modelling languages is the use of variable array lookups. Frequently CP models make use of the fact that we can build constraints where array lookups depend on variables (implemented in solvers by the `element` constraint). Consider an alternate knapsack model where we are restricted to take exactly k items:

```
int: k;
enum PRODUCT;
array[PRODUCT] of int: price;
array[PRODUCT] of int: profit;
array[1..k] of var PRODUCT: chosen;
constraint alldifferent(chosen);
constraint sum(i in 1..k)(price[chosen[i]]) <= budget;
solve maximize sum(i in 1..k)(profit[i]);
```

Note the second last line where we use a variable of enumerated type to look up the price of a product. This is a powerful feature of CP modelling languages.

In a language without strict type checking for enumerated types, this model will run and give seemingly meaningful answers (as long as there are more than k products). With strong type checking, a type error is reported, illustrating that the last line should read

```
solve maximize sum(i in 1..k)(profit[chosen[i]]);
```

4 Type Extensions

Enumerated types are a powerful modelling tool, and strict type checking has significant benefits, since the kind of errors that can arise without it may not necessarily be obvious to track down during the solving of the model. But together they may make it hard to express some reasonably common modelling patterns. One such modelling pattern is that we often want to reason about two or more sets of objects in the same way.

Example 1. Consider the usual objects for a vehicle routing problem:

```
enum CUSTOMER;  % set of customers to be served
enum TRUCK;     % set of trucks to deliver
```

The common *grand tour* modelling of such problems constructs a set of nodes: one node for each customer and two for each truck, a start node representing its leaving the depot, and an end node representing its return. Currently to represent such nodes we are forced to use integers, e.g.

```
set of int: NODE = 1..card(CUSTOMER)+2*card(TRUCK);
array[NODE] of var NODE:  next;   % next node after this one
array[NODE] of var TRUCK: truck;  % truck visiting node
```

This means we give up on type checking, risking the possibility of subtle modelling errors, particularly when doing arithmetic to access the truck nodes. □

In order to avoid moving to integers, we propose *enumerated type extensions*. They allow us to create new enumerated types by mapping existing enumerated types using type constructors and possibly adding new elements.

4.1 Syntax and Examples

Definition 2. *An enumerated type extension is defined by extending the syntax*

$$\langle enum \rangle \to \textbf{\textit{id}}\ \texttt{(}\ \langle enum \rangle\ \texttt{)}$$
$$\langle enum \rangle \to \langle enum \rangle\ \texttt{++}\ \langle enum \rangle$$

The first rule builds a new enumerated type from an existing one via a constructor function, while the second rule allows concatenation of enumerated types. □

Example 2. To express the node type using type extension we would write

```
enum NODE = C(CUSTOMER) ++ S(TRUCK) ++ E(TRUCK);
```

The new enumerated type has one element per customer and two per truck. We can access the names of the elements using the constructor functions, so e.g. the node for customer `c` is `C(c)` and the end node for truck `t` is `E(t)`. □

The order of the elements in the extension types is given by the order in the definition. In the example, customer nodes are before start nodes, which are before end nodes. The definition automatically creates the constructor function and its inverse, e.g. `C(.)` and $\texttt{C}^{-1}\texttt{(.)}$.[1] The inverse functions are partial. For example, $\texttt{C}^{-1}\texttt{(S(t))}$, which attempts to map the start node of truck `t` back to a customer, will become false in its enclosing Boolean context. We extend the constructor function to also work on sets of the base type, e.g. `C(CUSTOMER)` returns all the customer nodes.

Example 3. Given the `NODE` type defined in Example 2, we can set up the constraints on the trucks visiting each node as follows:

```
constraint forall(n in NODE diff E(TRUCK))
                 (truck[next[n]] = truck[n]);
constraint forall(t in TRUCK)(truck[S(t)] = t /\ truck[E(t)] = t);
```

That is, the truck visiting a node also visits its successor for all but the end nodes. And each truck visits its own start and end nodes. □

Note how the rules for type extension support concatenation of arbitrary enumerated types, not just the new constructor functions. This allows us to add "extra" elements to an enumerated type, as shown in the following example.

Example 4. A common modelling trick for vehicle routing problems where not every customer needs to be visited is to add a dummy truck, and all non-visited customers are "visited" by this truck. We can extended the enumerated type as

```
enum TRUCKX = T(TRUCK) ++ { DUMMYT };
```

The `NODE` type would then use `TRUCKX` instead of `TRUCK`. Now imagine we need to check that the individual trucks each visit between *mincust* and *maxcust* customers, and no more than *misscust* are not visited.

```
int: mincust;  % minimum customers visited by each truck
int: maxcust;  % maximum customers visited by each truck
int: misscust; % maximum missed customers
array[NODE] of var TRUCKX: truck; % truck or dummy visiting node
constraint global_cardinality_low_up([ truck[C(c)] | c in CUSTOMER],
                           TRUCKX,
                           [ mincust | t in TRUCK ] ++ [0],
                           [ maxcust | t in TRUCK ] ++ [misscust]);
```

[1] This can be written both in ASCII as `C^-1` or using the Unicode character for $^{-1}$.

The global cardinality constraint restricts the lower and upper bounds of the number of customers visited by each truck (including the dummy). □

Type extension is also useful for anonymous enumerated types, in particular if we have two or more anonymous enumerated types that we need to treat both separately and together.

Example 5. Consider a model for a graceful bipartite graph [5] defined as:

```
int: left;  enum LEFT = anon_enum(left);
int: right; enum RIGHT = anon_enum(right);
array[LEFT,RIGHT] of var bool: e;    % edges
var int: m = count(e);               % number of edges
enum NODE = L(LEFT) ++ R(RIGHT);
array[NODE] of var 0..left*right: label; % node label
constraint forall(n in NODE)(label[n] <= m);
constraint alldifferent(label);
constraint alldifferent_except_0([ e[l,r]*abs(label[L(l)] - label[R(r)])
                                 | l in LEFT, r in RIGHT ]);
```

The left and right nodes in the bipartite graph are separate anonymous enumerated types. The graph itself is represented by a 2D array of Booleans indicating which edges exist. We need to label nodes with different values from 0 to m where m is the number of edges. But the nodes are from two different classes, `LEFT` and `RIGHT`, so the `NODE` type is instrumental to defining the model. Finally each (existing) edge should be labelled with a different number from 1 to m. □

4.2 Pattern Matching and Range Notation

We extend the generator syntax of MiniZinc to include *pattern matching*, to make it easier to reason about different cases. In MiniZinc one can write a generator x `in` a where a is an array, so x takes the value of all elements of the array in turn. Once we have extended enumerated types it is worth extending this syntax to allow pattern matching: `P(`x`)` `in` a iterates through all elements of the form `P(`b`)` in a, setting pattern variable x to b for each such element.

Example 6. Consider a model for scheduling search and rescue teams of up to `size` members made up of humans, robots, and dogs. Each dog must be paired with their handler, and a robot requires a team member qualified to run them.

```
int size;
set of int: TEAM = 1..8;
enum PERSON;
enum DOG;
array[DOG] of PERSON: handler;
enum ROBOT;
array[PERSON] of set of ROBOT: skills;
enum MEMBER = P(PERSON) ++ D(DOG) ++ R(ROBOT) ++ { NOONE };
enum ZONE; % Zone to be searched
```

```
array[ZONE,TEAM] of var MEMBER: x;
% Each dog is paired with their handler
constraint forall(z in ZONE, D(d) in x[z,..])
                 (exists(t in TEAM)(x[z,t] = P(handler[d])));
% Each robot is in a team with the skills to run it
constraint forall(z in ZONE, R(r) in x[z,..])
                 (exists(P(p) in x[z,..])(r in skills[p]));
```

The constraints for dogs iterate over the zones and apply a constraint to team members matching the pattern $D(d)$. The robot constraints use pattern matching twice: to match the robots in a team, and to find the matching person. □

4.3 Implementing Enumerated Type Extension

Enumerated type extension allows for type safe construction of new types. Interestingly we can implement this feature entirely as syntactic sugar, i.e., by automatically rewriting extended enumerated types into standard MiniZinc.

In order to implement this feature, the MiniZinc lexer and parser need to be extended so that they recognise the new syntax. The type checking phase of the compiler is extended to introduce the new enumerated type, the constructor functions and inverse constructors. It makes use of the fact that we can always map enumerated types to integers.

Example 7. Consider the `NODE` type defined in Example 2. This is translated to a series of definitions:

```
int: nc = card(CUSTOMER);
int: nt = card(TRUCK);
enum NODE = anon_enum(nc + nt + nt);
function var NODE: C(var CUSTOMER: c) = to_enum(NODE,c);
function var NODE: S(var TRUCK: t) = to_enum(NODE,nc + t);
function var NODE: E(var TRUCK: t) = to_enum(NODE,nc + nt + t);
function var CUSTOMER: C⁻¹(var NODE: n) = to_enum(CUSTOMER,n);
function var TRUCK: S⁻¹(var NODE: n) = to_enum(TRUCK,n - nc);
function var TRUCK: E⁻¹(var NODE: n) = to_enum(TRUCK,n - nc - nt);
```

We introduce a new anonymous enumerated type of the right size. Each of the constructor functions coerces the original enumerated types to nodes using the `to_enum` function. Each of the inverse constructor functions performs the reverse coercion. Note that `to_enum`(E, i) is a partial function which is undefined if the integer second argument i is outside `1..card(E)`. This gives exactly the right behaviour for the partial inverse constructors. In the implementation we extend the constructors to also work on sets, and return sets, and generate specialized versions for when the input argument is fixed at compile time. □

Pattern matching expressions are again treated as syntactic sugar. The expression `[` $g(x)$ `|` `P(`x`) in` a `]` where x has type `T` is mapped to

`[ if` e `in P(T) then` g`(P⁻¹(e)) else <> endif | ` e `in` a `]`

The absent value <> acts as an identity element for the operator applied to the array, for more details see [8]. For special cases, in particular where a has par type (i.e., the test whether an element in a has the constructor P can be performed at compile time), we can avoid the creation of an array containing <> elements, but we leave out the details for brevity.

Example 8. The pattern matching in Example 6 is translated to

```
constraint forall(z in ZONE, i in TEAM, e = x[z,i])
      ( if e in D(DOG) then
          exists(t in TEAM)(x[z,t] = P(handler[D^-1(e)]))
        else true endif);
constraint forall(z in ZONE, e in x[z,..])
      ( if e in R(ROBOT) then
        exists(f in x[z,..])
              ( if f in P(PERSON) then R^-1(e) in skills[P^-1(f)]
                else false endif )
        else true endif);
```

where the compiler has replaced the absent value <> by the correct identity elements, `false` for the `exists`, and `true` for the two `forall` functions. □

5 Defaults

In MiniZinc, objects represented as enumerated types are usually implemented via arrays indexed by an object identifier. Type safety will check that we only access these arrays with the correct type. But this will often require us to guard the access to avoid undefinedness.

Example 9. In the vehicle routing problem, a critical part of the model is deciding arrival times at each node, based on some travel time matrix. Given data on customers and locations

```
enum LOCATION;  % set of locations of interest
array[CUSTOMER] of LOCATION: loc;    % location of customer
LOCATION: depot;                     % depot location
array[LOCATION,LOCATION] of int: tt; % travel time loc -> loc
array[CUSTOMER] of int: service;     % service time at customer
```

A model could decide the arrival time at each node as follows.

```
array[NODE] of var TIME: arrival;    % arrival time at node
constraint forall(t in TRUCK)(arrival[S(t)] = 0);  % start nodes
constraint forall(n in NODE diff E(TRUCK))(
  arrival[next[n]] >= arrival[n] +
  if n in C(CUSTOMER) then service[C^-1(n)] else 0 endif +
  tt[if n in C(CUSTOMER) then loc[C^-1(n)] else depot endif,
     if next[n] in C(CUSTOMER) then loc[C^-1(next[n])] else depot endif]
)
```

Note that we need to guard the lookup of the service time and location arrays to check that the node represents a customer, and then extract the customer from the node name. □

5.1 The `default` Operator

The guarding of data lookups, as well as the use of the inverse constructor to extract the subtype information is verbose. In order to shorten models and make them more readable we introduce *default* expressions into MiniZinc. Default expressions are not *directly related* to type extensions, rather they are a way of capturing undefinedness. However, they become particularly useful due to the addition of (partial) inverse enum constructors.

In MiniZinc expressions can be undefined, and take the value $\bot$, as a result of division by zero, or by accessing an array out of bounds. The undefined value percolates up the expression, making all enclosing expressions also undefined $\bot$ until a Boolean expression is reached where the undefinedness is interpreted as *false*; thus following the relational semantics treatment of undefinedness in modelling languages [3].

Many languages feature similar functionality. For example, C/C++ programmers may use a ternary operator to guard against `nullptr`. Haskell programmers would use the `maybe` function, and in Rust you might use `unwrap_or`.

Definition 3. *The default expression x `default` y takes the value x if x is defined (not equal to $\bot$) and y otherwise. If x and y are both $\bot$ the expression evaluates to $\bot$.* □

Example 10. With default expressions we can drastically shorten the arrival time reasoning shown in Example 9:

```
constraint forall(n in NODE diff E(TRUCK))(
   arrival[next[n]] >= arrival[n] +
   service[C^-1(n)] default 0 +
   tt[loc[C^-1(n)] default depot, loc[C^-1(next[n])] default depot]
)
```

The partial function `C^-1(n)` results in undefinedness when node `n` is not a customer node. This also makes the resulting array lookup undefined, which is then replaced by the default value. □

Default expressions are also useful for guarding other undefinedness behaviour. For example to calculate the minimum positive value occurring in a list, or return 0 if there are none, we can write

```
var int: minval = min([x | x in xs where x > 0]) default 0;
```

Defaults can also be useful for simplifying integer reasoning.

Example 11. A frequent idiom in constraint models over 2D representations of space is to use a matrix indexed by `ROW` and `COL`(umn). But then care has to be taken when indexing into the matrix. Imagine choosing k different positions in a matrix where the sum of (orthogonally) adjacent positions is non-negative. A model encoding this is

```
int: nrow; set of int: ROW = 1..nrow;
int: ncol; set of int: COL = 1..ncol;
```

```
array[ROW,COL] of int: m;  % given matrix
array[1..k] of var ROW: y; % row position chosen
array[1..k] of var COL: x; % col position chosen
constraint alldifferent([y[i]*ncol + x[i] | i in 1..k];
constraint forall(i in 1..k)
                 (sum(dr in -1..1, dc in -1..1 where abs(dr)+abs(dc) = 1)
                     (if y[i]+dr in ROW /\ x[i]+dc in COL
                      then m[y[i]+dr,x[i]+dc] else 0 endif) >= 0);
```

Notice that the model has to guard against the possibility that the position chosen is on one of the extreme rows or columns, e.g. `y[i] = 1`, since when `dr = -1` the lookup of `m` will fail and the relational semantics [3] will make the sum false. We can replace the sum if-then-else-endif expression simply by `m[y[i]+dr,x[i]+dc] default 0`. □

5.2 Implementing Defaults

A naive implementation would simply replace the expression `x default y` by

```
if defined(x) then x else y endif
```

given a suitable built-in function `defined`. Internally, the MiniZinc compiler already evaluates each expression into a pair of values: the result value of the expression, and a Boolean that signals whether the result is defined. We therefore chose to implement the `default` operator as a special built-in operation that can directly access the partiality component.

For the use case where the undefinedness arises from array index value out of bounds, the motivating case we consider, the MiniZinc compiler can choose to implement the default in a more efficient way than using if-then-else-endif.

For an expression $a[i]$ `default` y where i may possibly be outside the index set I of a we can build an extended array ax over the index set $lb(i)..ub(i)$, where $ax[i] = y$ for $i \notin I$, where $lb(i)$ ($ub(i)$) is the least (greatest) value in the declared domain of i.

We can extend this rewriting also to expressions of the form $a[f(i)]$ `default` y where f is a (possibly partial) function, by building an array ax over the index set $lb(i)..ub(i)$ where $ax[i] = y$ for $f(i) \notin I$ (including the case that $f(i)$ is not defined) and $ax[i] = a[f(i)]$ otherwise.

Example 12. This is particularly useful for undefinedness that results from the use of inverse constructors. Here we extend the array type to the full supertype `NODE`. Consider the arrival time constraint shown in Example 10. The automatic translation of defaults as extended arrays would then be

```
array[NODE] of int: servicex = array1d(NODE,
  [ if n in C(CUSTOMER) then service[C^-1(n)] else 0 endif | n in NODE]);
array[NODE] of LOCATION: locx = array1d(NODE,
  [ if n in C(CUSTOMER) then loc[C^-1(n)] else depot endif | n in NODE]);
constraint forall (n in NODE where not (n in E(TRUCK))) (
    arrival[next[n]] >= arrival[n] + servicex[n] +
                        tt[locx[n],locx[next[n]]]);
```

This is essentially equivalent to how an expert might write the model using integer indices. □

We can use the same approach for higher-dimensional arrays (as in Example 11). Note that if the bounds of the index variable i are substantially larger than the original index set of the array, the compilation approach may produce very large arrays (particularly for multi-dimensional arrays). Currently we limit the compilation of default expressions on arrays to no more than double the size of the original array, otherwise the if-then-else-endif interpretation is used.

6 Experiments

The first experiment is qualitative, examining how valuable the language extensions we propose here are likely to be. Considering all the models used in the MiniZinc challenge[2] as a representation of a broad range of constraint programming models, we examined each of the models to determine (a) if the model could be improved with (more) enumerated types; and (b) if the model could benefit from extensions and defaults. Note that some models used in the challenge were written before enumerated types were available in MiniZinc. In addition expert modellers (particularly those used to modelling directly for solvers) who submit models to the challenge often use integer domains even when an enumerated type might be suggested from the problem.

Of the 129 models used in the challenge over its history we find 15 that could make use of enumerated type extensions to improve type safety. Another 64 models could improve type safety simply by using enumerated types. Clearly the extensions we develop here are not restricted to a very special class of models.

As an example of a model that could be improved using enumerated type extensions we illustrate parts of the `freepizza` model. In the problem you must purchase a set of pizzas each with a given price, but you have vouchers that can be used, e.g. buy 2 get 1 free. A voucher is enabled by buying enough pizzas for it, then it can be used to get some free pizzas, but the free pizzas must always be no more expensive than the enabling bought pizzas. The key decisions in the original model are how you bought each pizza, expressed as follows.

```
int: m; % no of vouchers
set of int: VOUCHER = 1..m;
set of int: ASSIGN = -m .. m; % -i pizza is used to buy voucher i
                              %  i pizza is for free using voucher i
                              %  0 no voucher used on pizza
array[PIZZA] of var ASSIGN: how;
```

A key constraint in the model ensures that pizzas that enable a voucher are no less expensive than pizzas obtained for free:

```
constraint forall(p1, p2 in PIZZA)
                ((how[p1] < how[p2] /\ how[p1] = -how[p2])
                 -> price[p2] <= price[p1]);
```

[2] https://github.com/minizinc/minizinc-benchmarks

The ASSIGN set used in this model is an ideal case for an extended enumerated type. We can rewrite the model in a type-safe way as

```
int: m; % no of vouchers
enum VOUCHER = anon_enum(m);          % strong type check for VOUCHER
set of int: ASSIGN = Buy(VOUCHER) ++  % pizza is used to buy voucher v
                     { NOVOUCHER } ++ % no voucher used on pizza
                     Free(VOUCHER);   % pizza is for free using voucher v
array[PIZZA] of var ASSIGN: how;
```

The critical constraint is now simply

```
constraint forall(p1, p2 in PIZZA)
                 (Free(Buy⁻¹(how[p1])) = how[p2]
                  -> price[p2] <= price[p1]);
```

The partiality of the inverse constructors is used to trivially satisfy the implication. We would argue that the resulting model is far easier to understand than the original, and compared to the set `-m..m`, the extended type is self-documenting. Indeed a version of the original model has been used as a debugging exercise, since it is quite hard to reason about it. Note that because the original model uses negation to indicate that a voucher is bought, it represents those vouchers in the reverse order compared to the extended enum. The solver may therefore perform a different search, which results in different runtime behaviour (faster for some instances, slower for others). If negation `-v` in the original model is replaced by `v-m-1`, the two models behave identically.

Our second experiment demonstrates that translating array access expressions with defaults by extending the array with the default elements can lead to improvements in solving time. We ran a version of the capacitated vehicle routing problem from the MiniZinc benchmarks repository,[3] which we modified to use enumerated types and defaults. Table 1 shows the solving time and number of variables of defaults implemented as if-then-else-endif expressions[4] versus the extended arrays as explained in Example 12. For the experiments, we used the Chuffed solver with a timeout of 10 minutes, `A-n64-k9` and `B-n45-k5` data files, reduced to 8 and 9 customers to enable complete solving within the timeout. The results show an average improvement in solving time of 20%–30%, and a small reduction in the number of generated variables.

7 Related Work

Most programming languages support enumerated types in some form, it being a critical feature to avoid "magic constants". Enumerated type extension corresponds to using discriminated unions, for languages where those are available. No modelling language we are aware of except Zinc [7] supports such types, but

[3] https://github.com/minizinc/minizinc-benchmarks
[4] Compiled as described in [10].

Table 1: Solving times and number of generated variables for Chuffed on several CVRP instances with 8 and 9 customers, extended arrays (`x[y]`) versus if-then-else expressions (`i-t-e`).

Instance/Customer set	Solving time (sec)		No. of variables	
	`x[y]`	`i-t-e`	`x[y]`	`i-t-e`
B-n45-k5/1–8	1.724	2.060	39 794	40 122
B-n45-k5/9–16	1.776	2.217	39 722	40 050
A-n37-k5/1–8	6.997	8.251	38 372	38 700
A-n37-k5/17–24	9.432	10.726	37 894	38 222
B-n45-k5/25–32	9.545	12.340	41 262	41 590
A-n37-k5/9–16	10.115	14.727	33 104	33 432
B-n45-k5/17–24	13.290	24.691	43 556	43 884
A-n37-k5/25–32	20.608	35.316	33 834	34 162
B-n45-k5/1–9	31.266	43.143	47 683	48 065
B-n45-k5/19–27	125.936	177.199	54 928	55 183
B-n45-k5/28–36	159.009	209.935	50 427	50 809
A-n37-k5/1–9	174.749	223.807	46 499	46 881
B-n45-k5/10–18	169.007	229.181	45 787	46 169
A-n37-k5/19–27	189.714	265.302	42 611	42 993
A-n37-k5/10–18	254.691	346.922	47 647	48 029
A-n37-k5/28–36	262.204	366.466	42 347	42 729

Zinc does not support variables of such types, defeating one of the key purposes for introducing enumerated type extension.

AMPL [2] supports using sets of strings to define a form of enumerated types. Since the strings are only ever used as fixed parameters (there are no variables of type string) the language checks correct array lookups for arrays indexed by sets of strings during model compilation.

Similarly, OPL [11] does not support enumerated types, rather it supports the `string` data type, and the effect of enumerated types is mimicked by using sets of strings. Again since there are no variables of string type, the array index lookup for string indices is restricted to fixed parameters and checked during model compilation. Note that using strings to encode enumerated types has the advantage that one can simply build an array indexed by the union of two sets of strings, but this is not that helpful in the `NODE` example where we want to associate two nodes to each `TRUCK`. OPL does support arrays indexed by more complex types such as tuples which can significantly improve some models.

Essence [4] supports enumerated types that are very similar to MiniZinc's. They can be explicitly defined by sets of identifiers, in the model or the data, or defined as anonymous new types by size. Enumerated types can be used almost anywhere in the complex type language of Essence which includes parametric types for sets, multisets, functions, tuples, relations, partitions and matrices. Enumerated types support equality, ordering, and successor and predecessor functions. Essence is strongly typed, ensuring that all uses of enumerated types

are correct. Currently there is no way to coerce an enumerated value to an integer within Essence. In order to make use of global constraints on enumerated types the mapping of enumerated types to integers is performed during the translation of Essence to Essence' by Conjure. Because of this restriction there is no way to write an Essence model for the VRP using enumerated types, since one cannot associate enumerated types with (even integer) node values. This means an Essence model for VRP will be forced to use integers for all types `CUSTOMER`, `TRUCK` and `NODE`, thus losing strong type checking. We believe that the Essence type system can be extended to support the concepts presented here.

There are a number of constraint modelling languages with a focus on object orientation, where complex data is given as sets of objects as opposed to arrays indexed by enumerated types, and subclassing provides another approach to effectively reason about multiple different types of objects simultaneously.

In s-COMMA [1] one can define classes which include constraints across their fields, and (single inheritance) subclassing. Enumerated types are supported as base types (which cannot be subclasses). There are no variables that range across objects, meaning that the issues we address here don't arise.

ConfSolve [6] is an object-oriented modelling language aimed at specifying configuration problems. Again it supports enumerated types as base types that cannot be extended. The class system supports reference types which allow for powerful modelling of complicated relationships. This allows for similar kinds of subclass reasoning as extended enumerated types. It is not clear exactly how much type checking is applied to ConfSolve models. Interestingly the models are compiled to MiniZinc to actually run, essentially mapping object identifiers to integers and using arrays to represent fields and pointers to other objects.

8 Conclusion

Enumerated types are critical for type safety of models that manipulate objects. Type extension allows us to have the same safety properties for models that manipulate two sets of objects together, or need to extend a set of objects to define extreme cases. This is a frequent modelling pattern in complex constraint programming models. Hence we believe all CP modelling languages should support them. In this paper we show how they are implemented in MiniZinc, with enough detail so that other modelling language authors can translate the ideas to their own language.

We believe the use of enumerated types by modellers should be strongly encouraged, since we know that debugging models can be very challenging, and strong type checking of array access and function arguments can prevent very subtle errors when the model is solved.

Future work. The concept of enumerated type extension should generalise to tuple and record types, although the interactions of these types with arrays and decision variables are more difficult to handle in the compiler. Such an extension would make it much easier to interface MiniZinc models with object-oriented programming languages and data sources.

References

1. Chenouard, R., Granvilliers, L., Soto, R.: Model-driven constraint programming. Proceedings of the 10th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '08 (2008). https://doi.org/10.1145/1389449.1389479, http://dx.doi.org/10.1145/1389449.1389479
2. Fourer, R., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Duxbury (2002)
3. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In: Gent, I. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 5732, pp. 367–382. Springer (2009)
4. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
5. Golomb, S.W.: Graph Theory and Computing, chap. How to number a graph. Academic Press (1972)
6. Hewson, J.A.: Constraint-Based Specification for System Configuration. Ph.D. thesis, University of Edinburgh (2013)
7. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3), 229–267 (2008). https://doi.org/http://dx.doi.org/10.1007/s10601-008-9041-4
8. Mears, C., Schutt, A., Stuckey, P.J., Tack, G., Marriott, K., Wallace, M.: Modelling with option types in MiniZinc. In: Proceedings of the 11th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming. pp. 88–103. No. 8451 in LNCS, Springer (2014)
9. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer (2007)
10. Stuckey, P.J., Tack, G.: Compiling conditional constraints. In: de Givry, S., Schiex, T. (eds.) Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming. pp. 384–400. No. 11802 in LNCS (2019)
11. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press (1999)