# Two type extensions for the constraint modelling language MiniZinc

Rafael Caballero[a,**], Peter J. Stuckey[b,1,*], Antonio Tenorio-Fornés[a,*]

[a] *University Complutense of Madrid*
[b] *NICTA and the University of Melbourne*

## Abstract

In this paper we present two type extensions for the modelling language MiniZinc that allow the representation of some problems in a more natural way. The first proposal, called MiniZinc$^\star$, extends existing types with additional values. The user can specify both the extension of a predefined type with new values, and the behavior of the operations with relation to the new types. We illustrate the usage of MiniZinc$^\star$ to model SQL-like problems with integer variables extended with NULL values. The second extension, MiniZinc$^+$, introduces union types in the language. This allows defining recursive types such as trees, which are very useful for modelling problems that involve complex structures. A new *case* statement is introduced to select the different components of union type terms. The paper shows how a model defined using these extensions can be transformed into a MiniZinc model which is equivalent to the original model.

*Keywords:* Constraint Programming, NULL values, Union Types

[*]Corresponding author

[**]Principal corresponding author

[1]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

[2]Rafael Caballero partially supported by the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731), UCM grant GR3/14-910502, and the German Academic Exchange Service (DAAD)

## 1. Introduction

Constraint programming languages aim at providing mechanisms that allow the user to represent complex problems in a natural way. With that purpose, this paper presents two techniques for expressing constraints over extensions of the type system defined in the constraint modelling language MiniZinc [1].

In the first technique we allow extending existing types to include new values defined by the modeller. For example, within the new proposed framework, it is possible to extend the **int** predefined MiniZinc domain to support the representation of the value positive infinity. The new type `intE` is introduced by the reserved word **extended**:

```
extended intE = [] ++ int ++ [posInf];
```

where `posInf` is a new *extended constant*. Once a new extended type has been declared, the user can also define new operations as extensions of the predefined operations allowed by the language. For instance, in this example one could define the result of the addition of two `intE` variables `x` and `y`, either as `x+y` if both `x` and `y` are in the subtype **int**, or as `posInf` if at least one of the two values is `posInf` as in IEEE standard 754 [2].[3]

Apart from extended arithmetic, the extension of standard domains is an approach used in a multitude of disciplines, such as the design and testing of digital circuits [3], the representation of `null` values to express unknown data in database query languages such as SQL [4], or the many-valued logics [5]. All these problems can be successfully modeled in the language proposed in this paper, which we call MiniZinc$^\star$.

In the second technique we introduce the possibility of defining *union* types, also known as *sum* types [6]. An example is the distinction between a leaf and an interior node in a tree, where nodes can have children, but leaves do not. In the syntax of our proposal for introducing union types, which we call MiniZinc$^+$, a binary tree of integer numbers can be represented as:

[3]This standard is mainly devoted to the definition of Binary Floating-Point Arithmetic, but it also includes the definition of arithmetic extensions for finite numbers, infinities, and special "not a number" values (NaNs).

```
enum tree = { leaf(int),  node(int, tree, tree) };
var tree(3):t;
```

Although the union type `tree` represents general binary trees, in practice MiniZinc needs to know the size of the objects that define the model. This is needed because each variable of these new datatypes is translated into several MiniZinc variables that represent the atomic components of the structure. Since the set of MiniZinc variables must be determined at compile time, our translator needs to know the maximum number of variables that are necessary to mimic each MiniZinc$^+$ variable in advance. For this reason it is necessary to specify the maximum *level* of the data terms of the type when declaring the variables. For instance, the declaration **`var`** `tree(3):t;` indicates that `t` is a tree of maximum level 3. In our setting the level of any term is at least 0 (for constant terms), which means that in the example $t$ can only contain trees with a maximum height of 3.

We allow equality constraints for terms of union types, and introduce a new *case* statement for selecting the subterms of a term. In order to solve constraints over the new types, we present a source-to-source transformation from both models including the new types into MiniZinc. Source-to-source compilation is not the best choice from the point of view of performance, but we have chosen this technique for two main reasons:

1. The higher level provided by source-to-source transformations allows explaining clearly the transformation in terms of the well-known modelling language MiniZinc, and to prove the soundness of the approach.
2. MiniZinc generates FlatZinc, a front-end accepted for many different solvers (none of them including the features described in this paper to the best of our knowledge). Generating MiniZinc code means that any of those solvers can be used afterwards.

In the case of MiniZinc$^\star$, the transformation represents each extended decision variable as a pair of MiniZinc variables. The first variable contains a possible value of a standard type. The second variable contains a value in the extended type and also works as a switch that selects one of the two variables during the search. The transformation applies not only to constraint satisfaction problems, but also to optimization problems. In the case of union types, MiniZinc$^+$, each variable of a union type is represented by a set of variables that represent which constructor must be selected at

| Primitive | Purpose |
|---|---|
| **sv**([$e_1, \ldots, e_n$]) | Check that $e_1, \ldots, e_n$ correspond to standard values |
| **prdf**(op) | Call to the predefined operator `op` |
| **eq**(a,b) | Syntactic equality of `a` and `b`, used when redefining = |

Table 1: New Primitives introduced in MiniZinc$^\star$

each level, ensuring in this way the representation of any term of the type at the given level.

The next Section introduces MiniZinc$^\star$, its syntax based on MiniZinc with functions [7], and the transformation that converts MiniZinc$^\star$ models into MiniZinc models. Analogously, Section 3 introduces MiniZinc$^+$ and explains how models including extended types can be transformed into MiniZinc models. Section 4 discusses related work, while Section 5 presents the conclusions and discusses possible future work. Finally, Appendix A presents the soundness of the two approaches, while Appendix B discusses their possible combinations.

## 2. MiniZinc Type Extensions

### *2.1. Syntax*

MiniZinc is a medium-level constraint modelling language that allows the modeller to express constraint problems easily. In particular we take as starting point the version of MiniZinc with functions described in [7]. The grammar of MiniZinc$^\star$, the first MiniZinc extension proposed in this paper, is described in Figure 1. It corresponds basically to the grammar of MiniZinc, adding only the possibility of declaring new, extended types.
The non-terminal `model` is the start symbol of the grammar, `vId`, `fId`, `pId` and `tId` are identifiers for: parameters and variables, functions, predicates and new types, respectively. The terminal **`string`** represents an arbitrary string constant. The values $c_i$ represent new constant identifiers. The notation n$^{*[s]}$ / n$^{+[s]}$ indicates zero or more / one or more repetitions of the nonterminal "n" such that these repetitions are separated by string $s$. Boldface words are reserved words of the language.

The only difference of this grammar with respect to the standard MiniZinc with functions presented in [7] is the new nonterminal `typeE` and the

model $\longrightarrow$ typeE$^{*[;]}$;decl,$^{*[;]}$; assig$^{*[;]}$; pred$^{*[;]}$
; funct$^{*[;]}$;const$^{*[;]}$; solv; out;

type $\longrightarrow$ **int** | **bool** | **float** | tId | range

vtype $\longrightarrow$ type | **var** type

typeE $\longrightarrow$ **extended** tId =
[$c_{-n}, \ldots, c_{-1}$] ++type++ [$c_1, \ldots, c_m$]

exp $\longrightarrow$ vId | constant | vId[exp]
| arrexp[exp] | setexp | arrexp
| **if** exp **then** exp **else** exp **endif**
| pId(exp$^{*[,]}$) | fId(exp$^{*[,]}$)
| **let** {decl$^{*[,]}$ const$^{*[,]}$} **in** exp
| forall (arrexp)
| exists (arrexp)

arrexp $\longrightarrow$ [exp$^{*[,]}$]
| [exp | genvar$^{+[,]}$ **where** exp]

setexp $\longrightarrow$ { exp$^{*[,]}$ } | range
| {exp | genvar$^{+[,]}$ **where** exp}

genvar $\longrightarrow$ vId$^{+[,]}$ **in** setexp
| vId$^{+[,]}$ **in** arrexp

range $\longrightarrow$ exp .. exp

decl $\longrightarrow$ vtype : vId
| **array**[range] **of** vtype : vId
| **set of** type: vId
| **var set of** setexp: vId

assig $\longrightarrow$ vId = exp

const $\longrightarrow$ **constraint** exp

funct $\longrightarrow$ **function** decl (decl$^{*[,]}$) = exp

pred $\longrightarrow$ **predicate** pId(decl$^{*[,]}$) = exp

solv $\longrightarrow$ **solve satisfy** | **solve minimize** vId
| **solve maximize** vId

out $\longrightarrow$ **output** ([ sh$^{*[,]}$ ])

sh $\longrightarrow$ **show**(exp) | "string"

Figure 1: MiniZinc$^{\star}$ grammar

inclusion of type identifiers (`tId`) as possible types. Moreover, our setting includes the new built-in primitives listed in Table 1.

## *2.2. Example: Extending the Boolean type for a full adder combinational circuit*

Suppose that we wish to model combinational circuits with undefined (i.e. neither `true` nor `false`) signals [3]. Then, in our setting we can extend the standard MiniZinc Boolean type with a new constant `undef`. The definition in MiniZinc$^\star$ of the new type can be found in the first line of the model in Figure 2. Note that replacing `bEx` with **`bool`** in lines (3-6) and omitting lines (8-28) yields a standard MiniZinc model for this problem.

The model redefines the behavior of the Boolean connectives $\wedge$, $\vee$ and `xor` taking into account the new constant as indicated in the truth tables of Figure 3 (where 0 stands for `false`, 1 for `true` and $\perp$ stands for `undef`).

|  | **1** | **0** | $\perp$ |
|---|---|---|---|
| **1** | 1 | 1 | 1 |
| **0** | 1 | 0 | $\perp$ |
| $\perp$ | 1 | $\perp$ | $\perp$ |

(a) $\vee$

|  | **1** | **0** | $\perp$ |
|---|---|---|---|
| **1** | 1 | 0 | $\perp$ |
| **0** | 0 | 0 | 0 |
| $\perp$ | $\perp$ | 0 | $\perp$ |

(b) $\wedge$

|  | **1** | **0** | $\perp$ |
|---|---|---|---|
| **1** | 0 | 1 | $\perp$ |
| **0** | 1 | 0 | $\perp$ |
| $\perp$ | $\perp$ | $\perp$ | $\perp$ |

(c) xor

Figure 3: Truth tables including the undefined value

Observe that these tables correspond to both the *Kleene logic* and to the *Łukasiewicz logic* [5].

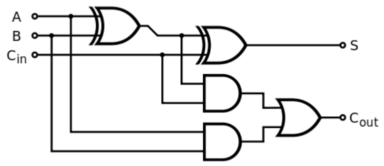
In our setting these tables are implemented by means of MiniZinc functions. For instance, the standard MiniZinc operator `xor` is redefined in MiniZinc$^\star$ as shown in lines (8-12) of Figure 2. The function first defines a local decision variable `c1`, which uses the predefined function **`sv`** in order to check if both parameters `a` and `b` contain standard values, that is, values different from `undef`. If this is the case (line 10), then the function returns the result of using the standard MiniZinc operator `xor`, which in our setting can always be accessed by using the wrapper **`prdf`** (standing for *predefined*).

Otherwise, if either `a` or `b` is `undef`, then the result is `undef` according to the table for extended `xor` of Figure 3. The schema of this function will be usual in all the `conservative` redefinitions of standard operators. The code for functions redefining $\wedge$ and $\vee$ is analogous.

```
extended bEx = bool ++ [undef];
int n;
array[1..n]   of var bEx: x;
array[1..n]   of var bEx: y;
array[1..n+1] of var bEx: s;
array[1..n+1] of var bEx: c;

function var bEx:xor(var bEx:a, var bEx:b) =
  let{var bEx:r, var bool:c1=sv([a,b]),
      constraint (c1 /\ (r = (a prdf(xor) b)))
                 \/ (not c1 /\ r=undef)
  } in r;

function var bEx:/\(var bEx:a, var bEx:b) =
 let{var bEx:r, var bool:c1=sv([a,b]),
     var bool:c2= (a=false \/ b=false),
     constraint (c1 /\ r = (a prdf(/\) b)) \/
                (not c1 /\ c2 /\ r=false) \/
                (not c1 /\ not c2 /\ r= undef)
     } in r;

function var bEx:\/(var bEx:a, var bEx:b) =
 let{var bEx:r, var bool:c1=sv([a,b]),
     var bool:c2= (a=true \/ b=true),
     constraint (c1 /\ r = (a prdf(\/) b)) \/
                (not c1 /\ c2 /\ r=true) \/
                (not c1 /\ not c2 /\ r=undef)
     } in r;

constraint c[1]=false /\ s[n+1]=c[n+1]
constraint forall([s[i]=x[i] xor y[i] xor
               c[i]|i in 1..n])
constraint forall([c[i+1]=(x[i] /\ y[i]) \/
        ((x[i] xor y[i]) /\ c[i])|i in 1..n]);
solve satisfy;
```

Figure 2: An $n$ bit full adder in MiniZinc$^\star$: $x + y = s$

Using these definitions we model the behavior of an $n$-bit adder digital circuit in lines (30-34). The basic piece of the circuit is the *full adder*:



which adds binary numbers and accounts for values carried in as well as out. The code of lines (30-34) employs $n$ full adders to obtain an $n$-bit adder. In particular, line (32) defines the output $s$ using two *xor* gates, while lines (33-34) model the carries employing two `and` and one `or` gates.

After transforming this model into a standard MiniZinc model, we can use MiniZinc to obtain solutions such as the following:[4]

$$\begin{array}{ccccccc}
\mathrm{x} & = & & 1 & 0 & \perp & 1 \\
\mathrm{y} & = & & 0 & 0 & \perp & 1 \\
\mathrm{c} & = & 0 & 0 & \perp & 1 & 0 \\
\hline
\mathrm{s} & = & 0 & 1 & \perp & \perp & 0
\end{array}$$

Observe that in the second position from the right the addition $\perp + \perp +1$ (1 is the carry from the rightmost position) yields $\perp$ in the result. In particular this means that the carry is undefined as well, and thus in the third position $0 + 0 + \perp$ produces the output $\perp$. However, in this case we can be sure that the carry is 0, and thus in the fourth position we have $1+0+0 = 1$ as output with carry 0 as last bit. Note also that the MiniZinc representation is by arrays where the least significant digit is the first position in the array, so the actual MiniZinc solutions is:

$$\begin{array}{lllllll}
\mathrm{x} & = [1, & \perp, & 0, & 1] & & \\
\mathrm{y} & = [1, & \perp, & 0, & 0] & & \\
\mathrm{c} & = [0, & 1, & \perp, & 0, & 0] \\
\hline
\mathrm{s} & = [0, & \perp, & \perp, & 1, & 0]
\end{array}$$

[4]The **`output`** sentence is omitted in Figure 2 for simplicity.

## *2.3. From MiniZinc$^\star$ to* MiniZinc

In this Section we present an automatic translation from MiniZinc$^\star$ to MiniZinc. Thanks to this translation, the models written in the extended setting can be solved using all the features (optimizations, different types of solvers, etc.) included in MiniZinc. The translation can be presented as a process in two phases:

1. First, functions, predicates and local declarations of variables are removed from the model.
2. Second, the resulting MiniZinc$^\star$ model, now containing neither functions nor local declarations, is translated into MiniZinc.

Observe that the first phase can be applied to both MiniZinc and MiniZinc$^\star$ indistinctly. In particular, the function elimination is done unrolling the function calls following ideas similar to those described in [7] (we assume in our setting the use of *total* functions), which simplifies the task. The elimination of constraints included in local declarations is managed using the relational semantics [8] of MiniZinc where these constraints "float" to the nearest enclosing Boolean context where they are added as a conjunct. Analogously, local variable declarations are converted to global variable declarations, see [9] for a more detailed discussion.

In the rest of the section we describe the second phase, which converts a MiniZinc$^\star$ model without functions and local declarations into a semantically equivalent MiniZinc model.

## *2.4. Transforming MiniZinc$^\star$ expressions*

In the case of MiniZinc$^\star$ expressions, the transformation is defined in terms of two auxiliary transformations, the first one representing the standard MiniZinc part of the expression (transformation $\tau_s(c)$), and the second one keeping a representation of the extended part (transformation $\tau_e(c)$).

## *2.5. Notation*

First we introduce some auxiliary notation:

We use $t$ for type identifiers (either standard as **`bool`**, **`int`** and **`float`** or extended such as `bEx`). Functions $st, et$ :: `Type` $\rightarrow$ `bool` take a type $t$ as parameter and return a boolean, indicating whether $t$ is either a standard ($st$) or an extended ($et$) type. Function $\texttt{ord}_t\texttt{(k)}$ :: `Constant` $\rightarrow$ `int` takes a constant `k` of type $t$ as parameter and return an integer that represents the *distance* to `k` from the base type following the textual order in its definition (the sub-index $t$ in `ord` is omitted when it is clear from the context). For instance, given the definition

```
extended int3 = [negInf]++int++[undef,posInf];
```

then:

- $\texttt{ord}_{int3}$`(negInf) = -1`
- $\texttt{ord}_{int3}$`(undef)= 1`
- $\texttt{ord}_{int3}$`(posInf) = 2`

For every constant $k$, $\texttt{ord}_t(k) \neq \texttt{0}$ iff $k$ is extended.

The function $\texttt{eRan}(t)$ (extended Range) takes an extended type $t$ as parameter and returns a MiniZinc range as follows: define a set $S$ as $S = \{\texttt{ord}_t(k) \mid k \in t\} \cup \{0\}$, then $\texttt{eRan}(t) = \min(S)\ ..\ \max(S)$. In the example of `int3` above: `eRan(int3) =` $-1\ ..\ 2$. We choose for each type $t$ a *default value* $k_{o(t)}$ which will be used in the representation of extended constants. The notation $o(t)$ refers to the base type of $t$ if it is extended, or to $t$ itself otherwise. Additionally, for each type $t$ we define a value $z_t$, which is 0 if $t$ is an atomic type, the array of $n$ zeros ($[0, \ldots, 0]$) if $t$ is an array of size $n$, the empty set ($\{\}$) if $t$ is a set, and the minimum value in the base type in the case of an integer subrange. In the rest of the paper we assume that MiniZinc$^\star$ models are well-typed following the type inference rules for MiniZinc which can be found in [10], and use the notation *type(e)* to refer to the type of $e$.

Next we explain the transformation of MiniZinc$^\star$ expressions, distinguishing between the different possibilities enunciated in the grammar (Section 2.1).

### *2.6. Identifiers, constants, array and set expressions*

*Base identifiers and constants.* The transformations $\tau_s$ (standard part) and $\tau_e$ (extended part) for identifiers and constants of base types are defined as follows:

| | $\tau_s$ | $\tau_e$ |
|---|---|---|
| Identifiers : $x$, $t = type(x)$ | | |
| $st(t)$ | $x$ | $z_t$ |
| $et(t)$ | $s(x)$ | $e(x)$ |
| Constants : $k$, $t = type(k)$ | | |
| $st(t)$ | $k$ | $z_t$ |
| $et(t)$ | $k_{o(t)}$ | $\texttt{ord}_t(k)$ |

Observe that here identifiers represent both decision variables and parameters. The table indicate that whenever $st(t)$ holds (that is, $t$ is a standard type), $\tau_s$ maps the identifier $x$ to itself, while $\tau_e$ maps $x$ to $z_t$, the arbitrary zero-value chosen for type $t$. For instance for **`var int`**`:x;` we have $\tau_s(x) = x$ while $\tau_e(x) = 0$.

If $et(t)$ then $x$ is an extended type identifier, and it is mapped to the associated new identifiers $s(x)$ of type $t$ and $e(x)$ of type `eRan`$(t)$.

Constants are mapped to themselves paired with $z_t$ if standard, or to the default constant from the underlying type and their order number if they are extended, new values. For instance the constant `negInf` of type `int3` verifies $\tau_s$`(negInf)` $= k_{\texttt{int}} = 0$, and $\tau_e$`(negInf)` $=$ $\texttt{ord}_{\texttt{int3}}$`(negInf)` $= -1$.

*Array expressions.* Array expressions of the form: $e = [e_1, \ldots, e_n]$ are transformed simply applying the transformations $\tau_s$, $\tau_e$ to each array element:

$$\tau_s(e) = [\tau_s(e_1), \ldots, \tau_s(e_n)] \qquad \tau_e(e) = [\tau_e(e_1), \ldots, \tau_e(e_n)]$$

For instance, consider the array expression $e$ defined as [*true,false,undef*]. Then, $\tau_s(e)$ = [*true,false,$\underline{\textit{false}}$*], and $\tau_e(e)$ = [*0,0,1*]. Observe that the underlined *false* corresponds to the arbitrary constant $k_{\texttt{bool}}$ chosen to replace *undef* and it is only used to keep the array with the same length and with the standard constants in the same positions.

*Array access.* An array indexing of the form $a[exp]$ with $type(a)$ = <*array of t*> is transformed as:

$$\tau_s(a[exp]) = \tau_s(a)[\tau_s(exp)] \qquad \tau_e(a[exp]) = \tau_e(a)[\tau_s(exp)]$$

We make use of the fact that MiniZinc arrays are always indexed by integers.

Consider the subexpression `c[1]` in line 30 of Figure 2. In this case, $type(c)$ = <*array of bEx*>, and thus *st(bEx)* is false while *et(bEx)* holds.

Therefore, $\tau_s$`(c[1])=cs[1]` , $\tau_e$`(c[1])=ce[1]`, assuming $s($`c`$)$ is defined as the new identifier `cs` and $e($`c`$)$ as `ce`.[5]

*Set expressions.* [6] Set expressions of the form $e = \{\ e_1,\ \ldots,\ e_n\ \}$ with $\mathit{type}(e_1) = \cdots = \mathit{type}(e_n) = t$ are transformed depending on the type $t$. If the type $t$ is standard, then the transformation just defines the extended part as the zero of the type, represented by the empty set, while the standard part is the set containing the standard components of the elements.

Otherwise, if $t$ represents an extended type, then the corresponding standard (extended) component must contain only those elements that actually take a standard (extended) value. This is achieved by using a set comprehension set with a guard of the form:

**where** $[\tau_e(e_1),\ \ldots,\ \tau_e(e_n)][\mathrm{i}] = 0$

for the case of the standard part (the extended part is obtained replacing $=0$ by $\neq 0$ in the expression above). Observe that for each $i$ this expression becomes true if the $i$-th element of the set $e$ takes a standard value (that is, its extended part is 0). Formally:

- if $\mathit{st}(t)$, then $\tau_s(e)$= $\{\ \tau_s(e_1),\ \ldots,\ \tau_s(e_n)\ \}$, and $\tau_e(e)$=$\{\}$.
- if $\mathit{et}(t)$, then the extended

$$\begin{aligned}
\tau_s(e) = \quad & \{[\tau_s(e_1),\ \ldots,\ \tau_s(e_n)][\mathrm{i}] \mid \mathrm{i\ in\ 1..n} \\
& \quad \textbf{where } [\tau_e(e_1),\ \ldots,\ \tau_e(e_n)][\mathrm{i}] = 0\ \} \\
\tau_e(e) = \quad & \{[\tau_e(e_1),\ \ldots,\ \tau_e(e_n)][\mathrm{i}] \mid \mathrm{i\ in\ 1..n} \\
& \quad \textbf{where } [\tau_e(e_1),\ \ldots,\ \tau_e(e_n)][\mathrm{i}] \neq 0\ \}
\end{aligned}$$

## *2.7. Array and set comprehensions*

Let $\langle$ `exp` | `genvars` **where** `cond` $\rangle$ be an array or set comprehension (with $\langle,\rangle$ representing either [,] or {,}). The translation of this expression consists of two phases. The first phase processes each generator $g$ in `genvars`. We use the notation $e[x \mapsto x']$ to indicate that all the occurrences of $x$ in $e$ must be replaced by $x'$.

- If $g \equiv$ `gId` **in** `genExp` with `genExp` a set or array of standard type, then apply the replacement `genvars[g` $\mapsto$ `gId` **in** $\tau_s$`(genExp)]`.

[5]For simplicity we use the suffixes `s` and `e` to generate new identifiers for the standard and extended parts of a construction in the rest of the paper.

[6]For set variables see Section 2.12.

- If $g$ is of the form `gId` **`in`** `arrayexp` and `arrayexp` is an array of extended type then:
  - Apply the replacement `genvars[g` $\mapsto$ `f` **`in`** `index_set(`$\tau_s$`(arrayexp))]`, where `f` is a fresh variable.
  - Apply the replacements `exp[gId` $\mapsto$ `arrayexp[f] ]` and `cond[gId` $\mapsto$ `arrayexp[f]]`
- If $g \equiv$ `gId` **`in`** `setexp` and `setexp` is a set of extended type then: Let fresh array expression `a` be

  `[`$ord_t^{-1}$`(x) | x` **`in`** $\tau_e$`(setexp)` **`where`** `x<0]++`

  `[x | x` **`in`** $\tau_s$`(setexp)]++`

  `[`$ord_t^{-1}$`(x) | x` **`in`** $\tau_e$`(setexp)` **`where`** `x>0]`.

  Then:
  - Apply the replacement `genvars[g`$\mapsto$ `f` **`in`** `index_set(`$\tau_s$`(a))]`, where `f` is a fresh variable.
  - Apply the replacements `exp[gId` $\mapsto$ `a[f] ]` and `cond[gId` $\mapsto$ `a[f] ]`

Let $\langle$`(exp')` $|$`genvars'` **`where`** `cond'` $\rangle$ be the result of applying this transformation to all the generators in the array/set comprehension. Then, the second phase of the translation is defined as:

- Array comprehensions:
$\tau_s =$ `[` $\tau_s$`(exp')` $|$ `genvars'` **`where`** $\tau_s$`(cond') ]`
$\tau_e =$ `[` $\tau_e$`(exp')` $|$ `genvars'` **`where`** $\tau_s$`(cond') ]`

- Set comprehensions:
$\tau_s =$ `{` $\tau_s$`(exp') | genvars'`
**`where`** $\tau_s$`(cond')` $\wedge$ $\tau_e$`(exp)=0 }`

$\tau_e =$ `{` $\tau_e$`(exp') | genvars'`
**`where`** $\tau_s$`(cond')` $\wedge$ $\tau_e$`(exp)` $\neq 0$ }

For example, let `intE` be the integer type extended with a new constant `posInf`, and consider the following expression:

```
e = [ y | x in [posInf, 4, 9, -1],
          y in {8, -1, 8, posInf}
          where x=y]
```

In order to simplify the presentation we use the notation `L` to represent the list `[posInf, 4, 9, -1]`, and `S` to represent the set `{8, -1, 8, posInf}`. Therefore, the array comprehension is represented as:

```
[y | x in L, y in S where x=y]
```

First we select the first generator `x in L`, choosing `i` as new variable and taking into account that $\tau_s$(`L`) = `[0, 4, 9, -1]`. Applying the replacements we obtain:

```
[y |  i in index_set([0,4,9,-1]), y in S
      where L[i]=y]
```

The second generator is `y in S`. Attending to the translation of set expressions we have:

$\tau_s(S) =$ `[ [8,-1,8,0][i]  | i in 1..4`
`                                 where [0,0,0,1] = 0]`

$\tau_e(S) =$ `[ [0,0,0,1][i]   | i in 1..4`
`                                 where [0,0,0,1]` $\neq$ `0]`

Then, the array expression $a$ is defined as:

`a  =  [`$\text{ord}_t^{-1}$`(x)  | x in `$\tau_e$`(S) where x `$<$` 0] ++`
`      [x | x in `$\tau_s$`(S)] ++`
`      [`$\text{ord}_t^{-1}$`(x)  | x in`$\tau_e$`(S) where x `$>$`0]`

Observe that during the evaluation of the model `a` will be evaluated to `[]++[-1,8]++[posInf] = [-1,8,posInf]`. The idea behind `a` is to obtain the list of elements in `S` without repetitions and respecting the order among elements. This mimics in MiniZinc⋆ the behaviour of MiniZinc where `[x | x in {3,4,5,3,4}]` is evaluated to `[3,4,5]`.
The translation proceeds by replacing the second generator by a new variable $j$, obtaining

```
[a[j] |  i in index_set([0,4,9,-1]),
         j in index_set(τs(a))
         where L[i]=a[j]]
```

Finally:

```
τs(e) = [τs(a[j])|  i in index_set([0,4,9,-1]),
                    j in index_set(τs(a))
                    where τs(L[i]=a[j]) ]
```

and

```
τe(e) = [τe(a[j])|  i in index_set([0,4,9,-1]),
                    j in index_set(τs(a))
                    where τs(L[i]=a[j]) ]
```

During the evaluation the system will obtain:

$\tau_s$`(e) = [0,-1]`, and $\tau_e$`(e) = [1,0]`,

which corresponds to the MiniZinc representation of the MiniZinc$^\star$ list `[posInf,-1]`.

*2.8. Conditional and logical expressions*

Expressions $e \equiv$ **if** c **then** e1 **else** e2 **endif** are transformed as:

```
τs(e) = if τs(c) then τs(e1) else τs(e2) endif
τe(e) = if τs(c) then τe(e1) else τe(e2) endif
```

Note: the `exists` and `forall` constructions are simply expanded to disjunctions and conjunctions respectively and then transformed.

*2.9. Predefined function and predicate calls*

We consider the following predefined function and predicate calls:

- $c \equiv$ **sv**`([exp`$_1$`,...,exp`$_n$`])`. The purpose of this Boolean function is to ensure that all the expressions correspond to standard values. Therefore: $\tau_s(c) =$ `(`$\tau_e$`(exp`$_1$`)=`$z_{t_1}$`)` $\wedge \cdots \wedge$ `(`$\tau_e$`(exp`$_n$`)=`$z_{o(t_n)}$`)`, with $z_{o(t_i)}$ the zero value associated to the type $t_i$ of expression `exp`$_i$.

- $c \equiv$ **prdf**`(f)(exp`$_1$`, ..., exp`$_n$`)`, or alternatively $c \equiv$ `exp`$_1$ **prdf**`(f)` `exp`$_2$, with `f` a predefined function or an infix operator. **prdf** indicates that this call corresponds to the predefined MiniZinc function/operator `f` even if it has been redefined by the user. Thus, $\tau_s$`(c)= f(`$\tau_s$`(exp`$_1$`), ..., `$\tau_s$`(exp`$_n$`))`, or $\tau_s$`(c)=`$\tau_s$`(exp`$_1$`) f `$\tau_s$`(exp`$_2$`)` if `f` is an infix operator, and $\tau_e$`(c)` $= z_t$ where $t$ is the output type of `f`. Thus, the user should ensure, usually by adding some constraints using **sv** that `exp`$_1$, ..., `exp`$_n$ can only correspond to standard values, otherwise the result of evaluating this function can be unsound.

- $c \equiv$ `(exp`$_1$ `= exp`$_2$`)`, assuming that = has not been redefined. Then: $\tau_s(c)$ is defined as

$$(\tau_s(\texttt{exp}_1) = \tau_s(\texttt{exp}_2) \wedge \tau_e(\texttt{exp}_1) = \tau_e(\texttt{exp}_2))$$

and $\tau_e(c)$ is defined as $z_{\texttt{bool}}$. The result of the comparison depends both on the standard and on the extended value. It is not enough to check only the standard part, because in case of two different extended constants `a`, `b` with base type $t$ we have $\tau_s(b) = \tau_s(a) = k_t$, but the result should be `false`. Analogously, the extended part is not enough because for instance considering the standard constants 3, 4, we have $\tau_e(3) = \tau_e(4) = z_{\texttt{bool}}$. The translation of `(exp`$_1$ $\neq$ `exp`$_2$`)` is simply `not(exp`$_1$ `= exp`$_2$`)`, followed by the translation of =.

- $c \equiv (e \texttt{ in } S)$, assuming that `in` has not been redefined. Then: $\tau_s(c) = (\tau_e(e) = 0 \wedge \tau_s(e) \texttt{ in } \tau_s(S)) \vee (\tau_e(e) \neq 0 \wedge \tau_e(e) \texttt{ in } \tau_e(S))$ and $\tau_e(c) = 0$. Other set operations such as `card`, `union` or `intersect` can be defined analogously.

This ends the transformation part for expressions. It only remains to define the transformation applied to top-level constructions.

## *2.10. Transforming MiniZinc$^\star$ models*

The transformation of a MiniZinc$^\star$ model $\mathcal{M}$, denoted by $(\mathcal{M})^{\mathcal{T}^\star}$ is obtained transforming each of these top-level constructions as described in this section.

## *2.11. Declarations of extended types*

The declarations of extended types are useful for obtaining the names of the new types, their base standard types, the names of the extended constants, and for generating the `ord` function described above. However, these declarations do not generate directly any code in the transformed MiniZinc model.

## *2.12. Declarations of variables and parameters*

If $c \equiv$ *decl* is a declaration of a variable or a parameter, then it is translated to MiniZinc as $(\texttt{decl})^{\mathcal{T}^\star}$ as defined by the following table:

| | $\mathcal{T}^\star$ |
|---|---|
| Var. or param. declarations: `[`**`var`**`]` $t$ : $x$, with $o(t) \in \{$**`int`**, **`float`**, **`bool`** $\}$ | |
| $st(t)$ | `[`**`var`**`]` $t$ : $x$ |
| $et(t)$ | `[`**`var`**`]` `o`$(t)$: $s(x)$; `[`**`var`**`]` `eRan`$(t)$: $e(x)$; $C_1$ |
| **`array`** [S] **`of`** [var] $t$: $a$ | |
| $st(t)$ | **`array`** [S] **`of`** [var] $t$: $a$; |
| $et(t)$ | **`array`** [S] **`of`** [var] $o(t)$: $s(a)$; **`array`** [S] **`of`** [var] `eRan`$(t)$: $e(a)$; $C_2$ |
| **`set of`** $t$: $x$ | |
| $st(t)$ | **`set of`** $t$: $x$; |
| $et(t)$ | **`set of`** `o(t)`: $s(x)$; **`set of`** `eRan`$(t)$ : $e(x)$ |
| **`var set of`** *setexp* : $x$, *type*(*setexp*) = <set of $t$ > | |
| $t$=**`int`** | **`var set of`** *setexp* : $x$ |
| $et(t)$ | **`var set of`** $\tau_s$(*setexp*) : $s(x)$; **`var set of`** $\tau_e$(*setexp*) : $e(x)$; |

with the constraints $C_1$ and $C_2$ defined as

$$C_1 \equiv \texttt{\textbf{constraint}}\ e(x) = z_{o(t)} \rightarrow s(x) = k_{o(t)}\texttt{;}$$

and

$$C_2 \equiv \texttt{\textbf{constraint}}\ \texttt{forall([}e(a[i]) \neq z_{o(t)} \rightarrow s(a[i]) = k_{o(t)}\texttt{| i \textbf{in} S]);}$$

The first column of the table distinguishes the different possible cases. The constraints $C_1$ and $C_2$ are introduced to avoid the repetition of equivalent solutions that is produced if the standard variables are not constrained. This is done, by ensuring that if the variable takes an extended value (extended part $\neq z_{o(t)}$), then the standard part of the variable takes some arbitrary value $k_{o(t)}$.

In our running example of Figure 2, the array `x` is transformed into:

```
array[1..n] of var bool: xs;
array[1..n] of var 0..1: xe;
constraint forall([ xe[i]!=0 -> xs[i]=false
                  |  i in 1..n ]);
```

assuming that `false` is the arbitrary constant $k_{\texttt{bool}}$.

## *2.13. Assignments and Constraints*

*Assignments* of the form $c \equiv vId = exp$, with $type(vId) = t$ are transformed as follows:

| | $\mathcal{T}^\star$ |
|---|---|
| *st(t)* | $vId = \tau_s(exp)$ |
| *et(t)* | $\tau_s(vId) = \tau_s(exp)$; $\tau_e(vId) = \tau_e(exp)$ |

Thus, the idea is to constrain the standard (respectively extended) part of the identifier to the standard (respectively extended) part of the expression.

*Constraints* have the form $c \equiv$ **constraint** *exp*`;`, where *exp* is a Boolean expression. In this case the transformation simply takes into account that the type of *exp* is standard, and therefore

$$c^{\mathcal{T}^\star} \equiv \texttt{constraint}\ \tau_s(exp)$$

## *2.14. Output Item*

The translation of an output item adds a new requirement, being able to print extended types. An expression of the form **show**`(exp)` must return a string representing the possibly extended expression `exp`. An extended type definition of the form

**extended** `tId [`$c_{-n}$, ..., $c_{-1}$`]++type++ [`$c_1$, ..., $c_m$`];`

creates an array of string `tnames`

```
array[eRan(tId)] of string: tnames =
       [c_{-n}, ..., c_{-1}, "dummy", c_1, ..., c_m];
```

and replaces each **show**`(e)` by

```
if(fix($\tau_e(e)$)==0)
then show($\tau_s(e)$)
else show(tnames[$\tau_e$(e)])
endif
```

For example `output [show(x)];` where `x` is of type `int3` creates

```
array[-1..2] of string: int3names =
     ["neginf","dummy","undef","posInf"];
output [ if (fix(xe) == 0) then show(xs)
         else show(int3names[xe]) endif ];
```

*2.15. Satisfaction and Optimization*

A *satisfaction problem* is encoded in MiniZinc$^\star$ using the solve item **solve satisfy**. In the translation to MiniZinc this is unchanged. However, MiniZinc also allows defining *optimization problems*, using the statements **solve minimize** $e$ or **solve maximize** $e$. In MiniZinc$^\star$ we also allow the optimization of expressions with extended range, extending implicitly the order $<$ to the new elements accordingly to their position with respect to the standard type in the definition of the type extension (see Section 2.4).

In standard MiniZinc, the optimization of an arithmetic expression is treated as the optimization of a variable constrained to be equal to the expression. Thus we consider goals either of the form *solve minimize y;* or *solve maximize y;* with $y$ a variable of some extended type $t$.

In order to compare values $k$ of extended types in the transformation we consider the lexicographical ordering over pairs of the form $(\tau_e(k), \tau_s(k))$. Let $a$ be the minimum base type value in $t$ if this exists, and $b$ be the maximum base type value in $t$ if this exists. If $a$ and/or $b$ do not exist, then $a = \min(\tau_s(y))$ and $b = \max(\tau_s(y))$. As a last resort, if we are to use a solver which artificially represents unbounded objects of the base type in a finite range $a..b$ we can use these values. Note that most finite domain solvers have this restriction. If we cannot determine either $a$ or $b$ then the optimization cannot be translated.[7] If $a$ and $b$ can be determined, we transform minimize/maximize $y$ as minimize/maximize $\tau_e(y) * (b - a + 1) + \tau_s(y)$.

[7]We are aiming to extend MiniZinc to directly handle lexicographic objectives, in which case this problem would disappear.

```
extended time = (0..23) ++ [oneDayOrMore];

function var time:+(var time:x, var time:y) =
 let {var time:r, var bool:c=sv([x,y]),
     constraint
      (c /\ x + y>23 /\ r=oneDayOrMore) \/
      (c /\ x + y<=23 /\ r=x prdf(+) y ) \/
      (not c /\ r=oneDayOrMore) }
      in r;

time: t1 = 5;
var time:t2;
var time:total = t1 + t2 + 21;
solve minimize total;
output(["Total=",show(total),
       " t2=",show(t2),"\n"]);
```

Figure 4: Modelling time with an extended value

For instance, the example in Figure 4 models the time required to perform some task. The time is measured in hours, from 0 to 23, plus a special value *oneDayOrMore*. The addition operator + is redefined accordingly, ensuring that if the sum of the two values exceeds 23 then the value *oneDayOrMore* is returned. For this type $a = 0$ and $b = 23$. This simple example indicates that task $t1$ requires 5 hours and that the goal is to minimize the function $t1 + t2 + 21$ represented by the decision variable *total*.

In the example, the sum of the values of the parameters exceed 23 hours, and therefore even assuming the minimum possible value for $c$ (which is 0), the expression takes the value *oneDayOrMore*. After transforming the model MiniZinc yields the expected values for variables *total* and *t2*:

```
Total=oneDayOrMore t2=0
```

## *2.16. Experimental Results*

This section presents a practical example of usage MiniZinc$^\star$ that has been used to check the feasibility of the proposal from the point of view of the implementation. The prototype implementation can be found at `https://gitorious.org/minizincplus`.

The tool *STCG* [11] generates MiniZinc models whose solutions constitute test-cases for testing SQL views.

Although realistic test-cases involve generating values for tables with several rows and queries relating different SQL views, we show here a very simple case of a test case for a SQL view defined as

```
create view V as
select *
from T
where (a != b or a != c) is null;
```

with $T$ a table defined as:

```
create table T (a int, b int, c int);
```

The condition indicates that the expression `(a != b or a != c)` must be evaluated to NULL. In fact NULL values are an important feature in the relational database model [12].

In order to generate the model that can represent this condition we extend the models including two new types: [8]

```
extended intN = [] ++ int ++ [NULL];
extended boolN = [] ++ bool ++ [NULLb];
```

And redefine the operators (=, !=, ∨, ∧) for integer and Boolean types extended with *NULL*:

```
function var boolN:'='(var intN:x, var intN:y)
  = let { var boolN: r,
          var bool: c = sv([x,y]),
          constraint (not(c) /\ eq(r,NULLb))
          \/ ( c /\ eq(r, (x prdf(=) y)))
  } in r;

function var boolN:'!='(var intN:x,var intN:y)
```

[8]This is also applicable to other domains allowed in SQL, but here we show these two types as an example.

```
  = let { var boolN: r,
          var bool: c = sv([x,y]),
          constraint (not(c) /\ eq(r,NULLb))
          \/ ( c /\ eq(r, (x prdf(!=) y)))
    } in r;

function var boolN:'/\'(var boolN:a,
                        var boolN:b) =
  let { var boolN:r,
        var bool:c1 = sv([a,b]),
        var bool:c2 = (eq(a,false)
                      \/ eq(b,false))),
    constraint (c1 /\ eq(r,(a prdf(/\) b)))
      \/ (not(c1) /\ c2 /\ eq(r,false))
      \/ (not(c1) /\ not(c2) /\ eq(r,NULLb))
    } in r;

function var boolN:'\/'(var boolN:a,
                        var boolN:b) =
  let { var boolN:r,
        var bool:c1 = sv([a,b]),
        var bool:c2 = (eq(a,true)
                      \/ eq(b,true))),
    constraint (c1 /\ eq(r,(a prdf(\/) b)))
      \/ (not(c1) /\ c2 /\ eq(r,true))
      \/ (not(c1) /\ not(c2) /\ eq(r,NULLb))
    } in r;
```

Observe that the redefinition of the operator = is specially intricate, analogously to the overloading of equality operators in case of object oriented languages (consider for instance the redefinition of `equals` in Java or C#). In our setting, the new built-in primitive **eq** represents the syntactic equality on expressions of extended type, that is the constraint that indicates that two expressions correspond to the same value. It behaves like the operator = if it has not been redefined. However, after redefining = the primitive **eq** becomes necessary for checking equality. Notice that **eq** cannot be substituted by **prdf**(=) because (a **prdf**(=) b) is only defined if a and b are extended expressions that are evaluated to standard values.

In the code above, after redefining the operator =:

- **eq**(r,NULLb) is a constraint which is satisfied only if r takes the value NULLb.
- r=NULLb is always evaluated to NULLb, because after the redefinition of = the equality of any value and NULLb is evaluated to NULLb.
- r **prdf**(=) NULLb is undefined because **prdf**(=) is not defined for extended values.

The previous MiniZinc* code allows the modeller to express the **where** condition of the SQL view to obtain a test case with a table containing a single row:

```
var intN: a;
var intN: b;
var intN: c;
constraint eq((a!=b \/ a!=c), nullb);
```

Of course, one could write the code directly in MiniZinc, but it leads to a more involved code. The hand-written code for this example would be something similar to:

```
var int: a;
var bool: aNull;
var int: b;
var bool: bNull;
var int: c;
var bool: cNull;

constraint aNull->(a=0);
constraint bNull->(b=0);
constraint cNull->(c=0);

constraint
    (aNull /\ bNull /\ cNull) \/
```

```
create table T (a int, b int, c int);
create view V as select * from T where a <> b or a <> c;
```

Figure 5: Sql Or: SQL example used in Table 2

```
    (cNull /\ not aNull /\ not bNull /\ a=b) \/
    (bNull /\ not aNull /\ not cNull /\ a=c);

solve satisfy;
```

The difference is clear even in this small example, and it becomes very noticeable in larger, more realistic, examples.

In order to check the efficiency of the proposal we have tried models produced by *STCG* for two SQL examples. The first example, which can be found in Figure 5 is a simple query over just one table. In the second example (Figure 6) the table *board* represents the position (x, y) and player (id) of pieces of a game in a two dimensional grid, and the view *checked* shows the players with at least one piece threatened (in the same row or column) by another player piece.

Table 2 shows the data obtained with our current implementation. The meaning of the columns of this table is the following:

- *Sql Or* and *Board*: Name of two standard MiniZinc models generated automatically by *STCG* in order to obtain test-cases for the SQL examples of Figures 5 and 6.

  This MiniZinc code generated by *STCG* does not take into account the possibility of obtaining *null* values in the SQL columns (represented in the model by integer MiniZinc variables).

- *Sql Or*$^T$ (respectively *Board*$^T$) are MiniZinc models obtained from *Sql Or* (respectively *Board*) as follows:

  1. Introduce in each model extended types:

     ```
     extended intN = [] ++ int ++ [NULL];
     extended boolN = [] ++ bool ++ [NULLb];
     ```

```
create table player (id int, primary key(id));

create table board (x int, y int, id int,
  primary key(x,y),
  foreign key (id) references player(id));

create view nowPlaying(id) as
  select p.id
  from player p
  where exists (select b.id from board b where b.id=p.id)
     ;

create view checked(id) as
  select p.id
  from player p
  where exists (select n.id from nowPlaying n where n.id
     = p.id)
    and not exists (select b1.id from board b1
                    where b1.id = p.id and
                          not exists
                         (select b2.id from board b2
                          where (b2.x - b1.x) * (b2.y-b1.
                             y)=0 and
                                (b1.id <> b2.id)));
```

Figure 6: Board: SQL example used in Table 2

| Model | *Sql Or* | *Sql Or*$^T$ | *Board* | *Board*$^T$ |
|---|---|---|---|---|
| var. decl. | 3 | 6 | 8 | 16 |
| var. flat. | 5 | 32 | 47 | 7291 |
| funct. calls | 4 | 264 | 419 | 3556 |
| size (KB) | 0.5 | 5.5 | 13.2 | 528.6 |
| transf. time |  | 5 |  | 530 |
| solve time | 0.2 | 0.2 | 0.3 | 2.5 |

Table 2: Experimental data for two models generated by STCG

Then, change the types of the MiniZinc decision variables used in the test case to the new types, and introduce the code of the functions redefining the logic operators taking into account the new *null* value as explained above. Let *Sql Or*$^\star$ (respectively *Board*$^\star$) be the MiniZinc$^\star$ models obtained in this way.

2. The MiniZinc$^\star$ models *Sql Or*$^\star$ and *Board*$^\star$ are transformed as explained in this section. The result are two MiniZinc models called in the Table *Sql Or*$^T$ and *Board*$^T$, respectively.

Thus, *Sql Or* and *Board* are two models that represent two examples of use of MiniZinc for obtaining SQL test-cases without taking into account *null* values, while *Sql Or*$^T$ and *Board*$^T$ represent the same problem but now taking into accout *null* values.

The rows of the table contain:

- *var. decl.*: Number of declared variables in the model. For instance in the case *Board* in the MiniZinc model produced by *STCG* for the second example are transformed into 16 variables in the model when considering NULL values.

- *var. flat.*: Number of variables in the FlatZinc transformation of the model. The flat version of the model shows better the amount of variables involved in the model. The flattening of calls to functions is the main reason of the increment in the number of variables.

- *funct. calls*: The number of function calls included in the code, including calls to the predefined operators $\{=,!=,\vee,\wedge\}$. For instance in the first example *STCG* generates a model including only 4 calls. After extending the model to MiniZinc$^{\star}$ to support NULLs and applying the transformation to obtain the equivalent MiniZinc model we obtain a model with 264 function calls.

- *size*: The size in Kbytes of the models. It can be seen that the size increases dramatically after the transformation.
- *transf. time*: Time required by the transformation in milliseconds. In the more complex example of *Board* about 3 seconds are required by our prototype to convert the MiniZinc$^{\star}$ model into a MiniZinc model.
- *solve time*: The time required by the MiniZinc solver to obtain the first answer in milliseconds.[9]

The implemented version of the tool does not perform common subexpression elimination. This not only affects the solving performance [7] but also the number of function calls and the size of the model. Despite this increment in size, number of variables and function calls, the experimental results show that the theoretical proposal can be used in practice.

The prototype is available at
`git@gitorious.org:minizincplus/minizincplus.git`

## 3. Union Types

This section is devoted to the second extension we propose, which consists on the possibility of introducing *union types* in MiniZinc. This variant of MiniZinc is denoted by MiniZinc$^{+}$ in this paper. The new types are introduced via **`enum`** declarations similar to those allowed in the Zinc modelling language [14], but with the noticeable difference of allowing recursive types. This facilitates modelling problems associated to recursive structures such as trees. One difference of MiniZinc$^{+}$ with respect to union types in functional languages is that MiniZinc models must represent finite sets of values of bounded size. To preserve this property the modeller must introduce an upper bound or *level* that limits the size of the structures.

In order to define and use the new datatypes we allow the following novelties in MiniZinc$^{+}$ models:

[9]Data from Gecode [13] FlatZinc solver statistical information.

- A syntax extension for defining union types.
- A new **`case`** statement for referring to subterms of the constructed terms.
- A redefinition of the equality constraint for dealing with the new types.
- Recursive predicates are now allowed (they are not allowed in standard MiniZinc).

We also introduce a transformation that converts the new MiniZinc$^+$ models into standard MiniZinc models, so they can be solved using any off-the-shelf constraint solver employed usually with MiniZinc. While Zinc supported the translation of tuple and record types to base solver types [15], it never supported the translation of union types, and indeed the approach of Zinc cannot be extended to recursive union types, since it occurs at compile time, before the data is known.

*3.1. Syntax*

The syntax of this feature is the same as described in Section 2.1 after replacing the definition of the extended types by the new union types with the following syntax:

$$\texttt{\textbf{enum}}\ \mathrm{T} = \{\, c_1(t^1_1, \cdots, t^1_{m_1}),\ \cdots\ ,\ c_n(t^n_1, \cdots, t^n_{m_n})\}$$

with $T$, is a new type name $t^i_j$ are already existing type names, and $c_i$ identifiers representing constructor names. In our setting the only expressions of union type $T$ allowed are variables of the type, and constructor terms (*cterms* in the following) of the form $c_i\ e_1 \ldots e_n$ with $1 \leq i \leq n$ and $e_j$ expressions of type $t^i_j$ for $j = 1 \ldots m_i$. An important concept is the *level* of a term:

1. The level of terms of standard MiniZinc types such as **`int`** is always 0.
2. The level of a completely defined term (term without variables), corresponds to the height of the syntax-tree representing the structure of the term.
3. In the case of variables of union type, the user must indicate in the variable declaration the level of the associated term, that is the maximum level allowed for terms represented by this variable.

4. The level of a constructed term that includes variables is

$$\texttt{level}(c_i(e_1,\dots,e_n)) = 1 + \texttt{max}(\texttt{level}(e_1),\dots,\texttt{level}(e_n))$$

An associated concept is the *path* of a subterm $x'$ of a completely defined term $x$, $\texttt{path}(x, x')$. If $x' = x$ then $\texttt{path}(x, x') = \varepsilon$. If $x' \neq x$, let $x''$ be the inner subterm of $x$ that contains $x'$, $x'' = c_i(\dots, x_{j-1}, x', x_{j-1}, \dots)$. Then, $\texttt{path}(x, x') = \texttt{path}(x, x'').j$. The sets of paths of a term $x$, $\texttt{spath}(x)$ is the set containing all the paths of subterms in $x$.

The same concept can be extended to variables of type $T$ and level $l$: their sets of paths are defined as the union of all the sets of positions of completely defined terms of type $T$ at level $k$ for $k \leq l$, and to non-completely defined terms of type $T$ (that is, terms including variables at some positions).

Observe that paths play a rôle similar to that of positions in term rewriting. Similarly to the case of positions we assume the notation $q \leq p$ for indicating that the path $q$ is a prefix of path $p$. We assume that $p.\varepsilon.q$ and $p.q$ represent the same position, and in particular write $p$ instead of $\varepsilon.p$.

### *3.2. Example*

Figure 7 models the following well-known problem: given an initial set of numbers $S$ and a number $N$, try to obtain an arithmetic expression that yields $N$ from the values in $S$ by using the four basic arithmetic operations: addition, subtraction, multiplication and division. Divisions can only be applied to operands divisible without remainder. Parentheses are allowed, and we can suppose that any value in $S$ can be used any number of times (including 0), but that the size of the expression has some arbitrary upper bound. For instance, considering the set $S = \{3, 14, 32\}$ and $N = 7$, if we limit the numbers in the expressions to a maximum of 4, we have the following solution (among others): $((32 + 3) - 14)/3 = 7$.

Modelling this problem in MiniZinc is not straightforward, since the natural representation of arithmetic expressions is as arithmetic trees with operations in the internal nodes and numbers at the leaves.

In our extension the representation is quite simple. First the input parameters are declared: the number to obtain (7), the maximum amount of numbers (4), and the set $S$ of input numbers. Then, we declare the possible operators, and the type `tree` representing the expression. Both are union types. A variable $t$ of the union type *tree* is then declared. The declaration includes between parentheses the maximum depth of the term of type *tree*

```
int:N=7;
int:level = 4;
set of int: S = {3,14,32};
enum op = {add , subst , multi , divi};

enum tree = { leaf(int), node(op, tree, tree)};

var tree(level):t;

predicate validTree(var tree:t, var set of int: S) =
   case t of
      leaf(x)       --> x in S;
      node(o,t1,t2) --> validTree(t1,S) /\ validTree(t2,S);
   endcase;

predicate value(var tree:t, var int:v) =
case t of
   leaf(x) --> v=x;
   node(o,t1,t2) --> let {var 0..1000:v1, var 0..1000:v2} in
                     (value(t1,v1) /\ value(t2,v2) /\
                     appOp(o,v1,v2,v));
endcase;

predicate appOp(var o:op,var int:v1,var int:v2,var int:v)=
   case o of
      add --> v = v1+v2;
      subst --> v = v1-v2;
      multi --> v = v1*v2;
      divi --> v = v1 div v2 /\ v1 mod v2 = 0;
   endcase;

constraint validTree(t,S);
constraint value(t,N);

solve satisfy;
output([show(t)]);
```

Figure 7: Numeric expression producing a given value

that $t$ can contain. In this case it is straightforward to check that the the maximum amount of numbers allowed in the expression is an upper bound of the depth of the tree. We require that the tree is valid in the sense that the leaves are elements of $S$, and that the result of evaluating the tree is $N$. In the predicate `value` two local variables are introduced in order to compute partial results with a fixed range (declaring them as integers causes overflow in MiniZinc). Observe that the three predicates employ the **`case`** statement for distinguishing the possible forms of types/operators. Finally we solve the model (**`solve satisfy`**), and display the result **`output`**`[`**`show`**`(t)]`.

In the next subsections we explain the details of the proposed extension together with its transformation into standard MiniZinc.

## *3.3. Variable declarations*

Each variable in MiniZinc$^+$ is transformed into MiniZinc as a set of variables and one constraint.

In the set of variables there is a distinguished variable, called the *selector* variable, that indicates the outer constructor in the value that the variable can take.

More formally, given a union type

$$\texttt{enum}\ \mathrm{T} = \{\, c_1(t^1_1, \cdots, t^1_{m_1}),\ \cdots,\ c_n(t^n_1, \cdots, t^n_{m_n})\}$$

We define the transformation of variable $x$ of union type $T$ at level $l$ (represented by the notation $(x : T, l)^{\mathcal{T}^+}$) as a tuple $(V, C)$, with $V$ a set of variables with their types, and $C$ a constraint, defined as follows:

- if $l\ =\ 0$:
$$V = \{x : 1..n\}\ \ \texttt{and}\ \ C = \bigwedge_{\substack{i\ \in\ 1..n \\ m_i\ >\ 0}} x \neq i$$

- if $l\ >\ 0$. For $i = 1 \ldots n$, $j = 1 \ldots m_i$ let $x_{ij}$ be new variable names, and let $(V_{ij}, C_{ij})$ be defined as:
  1. $(V_{ij}, C_{ij}) = (\{x_{ij} : t^i_j\}, \texttt{true})$ if $t^i_j$ is a standard MiniZinc type.
  2. $(V_{ij}, C_{ij}) = (x_{ij} : t^i_j, l-1)^{\mathcal{T}^+}$ otherwise.

  Then:
$$V = \{x : 1..n\}\ \cup\ \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} V_{ij}$$

and

$$C = \bigwedge_{i=1}^{n} \{(x \neq i \rightarrow \bigwedge_{j=1}^{m_i} ( \bigwedge_{v:t' \in V_{ij}} v = z_{t'})) \wedge (x = i \rightarrow \bigwedge_{j=1}^{m_i} C_{ij})\}$$

As the definition shows, the selector variable is always transformed into a variable with the same name and range `1..n`, where $n$ is the number of constructors defining the type. When the level is 0 we add constraints to ensure that the selector variable can only take the index value of a constructor without arguments, which is the only 0-level term allowed. In the case of a level greater than 0 we collect all the variables obtained in the recursive calls plus the selector. Moreover, we include a constraint which is the conjunction of the constraints for each value $i = 1 \ldots n$. For each $i$ the generated constraint consists of two parts:

1. If the selector takes a value different from $i$ then ensure that all the variables associated to the $i\,th$ constructor take an arbitrary value $z_{t'}$ ($t'$ the type of the corresponding variable). This is done because if these variables were left unbound they would take all the values in their domain and produce repeated solutions.
2. If the selector variable takes the value $i$ in a solution then it must verify also all the constraints associated to the variables associated to this constructor.

For instance, the fragment of code

```
enum op = {sum , minus , prod , div};
enum tree  { leaf(int),  node(op, tree, tree) };
var tree(2):t;
```

is transformed into standard MiniZinc as:

```
var 1..2: t;
var int: t_1_1;
var 1..4: t_2_1;
var 1..2: t_2_2;
var int: t_2_2_1_1;
var 1..4: t_2_2_2_1;
var 1..2: t_2_2_2_2;
var 1..2: t_2_2_2_3;
var 1..2: t_2_3;
var int: t_2_3_1_1;
var 1..4: t_2_3_2_1;
var 1..2: t_2_3_2_2;
var 1..2: t_2_3_2_3;

constraint
((t!= 1)->(t_1_1= 0)) /\
((t!= 2)->(t_2_1= 1 /\
           t_2_2= 1 /\ t_2_2_1_1= 0 /\ t_2_2_2_1= 1
                    /\ t_2_2_2_2= 1 /\ t_2_2_2_3= 1 /\
           t_2_3= 1 /\ t_2_3_1_1= 0 /\ t_2_3_2_1= 1 /\
                       t_2_3_2_2= 1 /\ t_2_3_2_3= 1)) /\
((t= 2) ->( ((t_2_2!= 1)->(t_2_2_1_1= 0)) /\
          ((t_2_2!= 2)->(t_2_2_2_1= 1 /\ t_2_2_2_2= 1 /\
                         t_2_2_2_3= 1)) /\
          ((t_2_2= 2) ->(t_2_2_2_2!= 1 /\ t_2_2_2_2!= 2 /\
                         t_2_2_2_3!= 1 /\ t_2_2_2_3!= 2)) /\
          ((t_2_3!= 1)->(t_2_3_1_1= 0)) /\
          ((t_2_3!= 2)->(t_2_3_2_1= 1 /\
                         t_2_3_2_2= 1 /\ t_2_3_2_3= 1)) /\
          ((t_2_3= 2) ->(t_2_3_2_2!= 1 /\ t_2_3_2_2!= 2 /\
                         t_2_3_2_3!= 1 /\ t_2_3_2_3!= 2))));
```

Figure 8 can help to understand the meaning of the variable names. For instance, the variable $t$ of type `tree` can take either the value 1 indicating that $t$ is a `leaf`, or 2 indicating that $t$ is of the form `node(...)`. The only variable associated to the `leaf` case is `t_1_1` that represents the number stored in this leaf. The constraint `(t`$\neq$`1)` $\rightarrow$ `(t_1_1= 0)` indicates that this variable is arbitrarily set to 0 when $t$ is not 1. On the contrary, if $t$ is 1 (that is, $t \neq 2$), then all the rest of the variables are constrained to take arbitrary values. Finally, if $t = 2$ we need more variables for the arguments of `node`. This first one, **`var`** `1..4: t_2_1;` represents the operation stored at the node. The other two variables are **`var`** `1..2: t_2_2;` and **`var`** `1..2: t_2_3;` are selectors the left and the right child trees, respectively. The meaning of the rest of the variables can be obtained applying recursively Figure 8. This leads to a more readable format for variables. For instance, `t_2_3_2_2` can be written as $t_{\texttt{node(\_,\_,node(\_,*,\_))}}$, indicating that this variable contains the value `x` such that `t = node(_, _, node(_,x,_))`. This notation is used

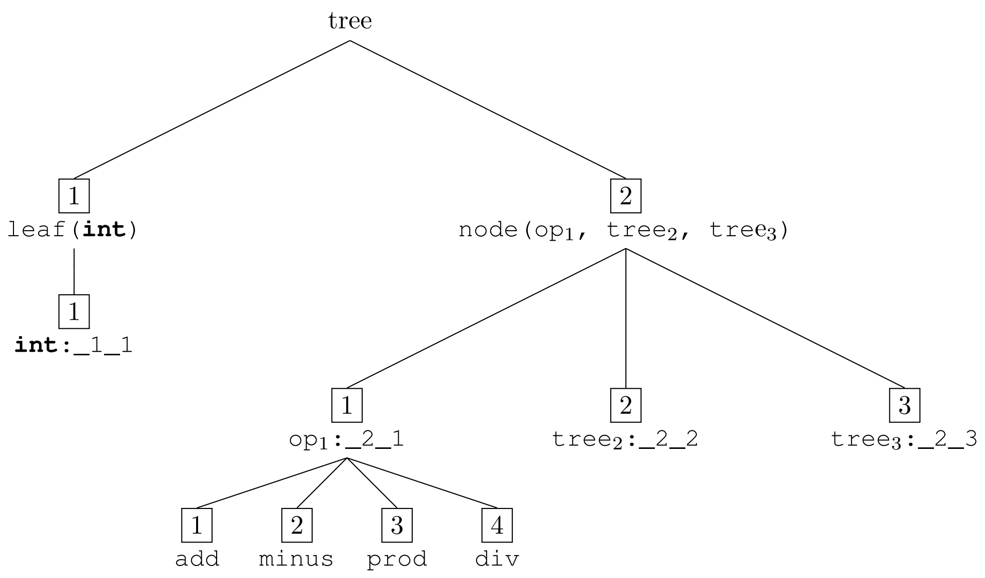


Figure 8: MiniZinc variables associated to a MiniZinc$^+$ variables of type `tree`

in the rest of the section when convenient. As a first example, we consider again part of the constraints associated to the transformation of variable `t` displayed above:

$$\begin{array}{ll}
((t \neq \texttt{leaf}) \rightarrow & \cdots \wedge \\
((t \neq \texttt{node}) \rightarrow & \ldots \wedge \\
((t = \texttt{node}) \rightarrow & (((t_{\texttt{node}(\_,*,\_)} \neq \texttt{leaf}) \rightarrow \ldots) \wedge \\
 & ((t_{\texttt{node}(\_,*,\_)} \neq \texttt{node}) \rightarrow \ldots) \wedge \\
 & ((t_{\texttt{node}(\_,*,\_)} = \texttt{node}) \rightarrow \\
 & \qquad (t_{\texttt{node}(\_,\texttt{node}(\_,*,\_),\_))} \neq \texttt{leaf} \wedge \\
 & \qquad t_{\texttt{node}(\_,\texttt{node}(\_,*,\_),\_))} \neq \texttt{node} \wedge \\
 & \qquad t_{\texttt{node}(\_,\texttt{node}(\_,\_,*),\_))} \neq \texttt{leaf} \wedge \\
 & \qquad t_{\texttt{node}(\_,\texttt{node}(\_,\_,*),\_))} \neq \texttt{node})) \ \wedge \ \ldots)
\end{array}$$

Observe in particular the part devoted to ensuring that the tree level is satisfied by any solution of the model:

$$\begin{array}{ll}
((t = \texttt{node}) \rightarrow & \ldots \\
 & ((t_{\texttt{node}(\_,*,\_)} = \texttt{node}) \rightarrow \\
 & \qquad (t_{\texttt{node}(\_,\texttt{node}(\_,*,\_),\_))} \neq \texttt{leaf} \wedge \\
 & \qquad t_{\texttt{node}(\_,\texttt{node}(\_,*,\_),\_))} \neq \texttt{node} \ \wedge \ \ldots
\end{array}$$

the conjunction $t_{\texttt{node(\_,node(\_,*,\_),\_)}} \neq$ `leaf` $\wedge$ $t_{\texttt{node(\_,node(\_,*,\_),\_)}} \neq$ `node` indicates that the variable $x$ in a tree of the form `node(_,node(_,x,_),_)` can be neither a `leaf` nor a `node`. But any tree must take one of the two forms. Thus, in order to make the constraint true, either of the outer implication conditions $t_{\texttt{node(\_,*,\_)}} =$ `node` or $t =$ `node` must be false. If $t_{\texttt{node(\_,*,\_)}} =$ `node` is false this means that the form `node(_,node(...),_)` is not allowed for variable `t`, which is logical since we are limiting the maximum level of `t` to 2. If the outer implication condition $t =$ `node` does not hold the tree corresponds to a single leaf, and this also ensures that the level constraint is kept. Another part of the constraint, not displayed here for the sake of space, indicates that the form `node(_,_,node(...))` is also forbidden.

This small example shows the main drawback of our approach: the great amount of variables necessary for representing the union type variables which grows exponentially with the level. In this example, with level=2, 13 variables have been generated. With level=4, as in the example of Figure 7, a total of 63 variables are produced, and with level 8 a total of 1021 variables are necessary. Of course some of the variables can be eliminated analyzing the associated constraints (in fact all the variables whose suffix considered as a path is not in `path`$(x)$), but this only affects the variables of the last level and does not prevent the exponential growth. In any case, the generated variables are defined in small ranges and related by equality and inequality constraints, which are good properties for finite domain solvers.

## *3.4. Equality*

We define the transformation of the equality constraint between two union type expressions $e_1, e_2$ introducing the following auxiliary notation:

- Let $x$ be a variable of type $T$ defined at level $l$, and $p$ a path in `spath`$(x)$, then $[x]_p = x_p$, with $x_p$ the MiniZinc variable obtained in the transformation of $x$ that corresponds to path $p$.

  For instance, given the variable declaration **`var`** `tree(2):t;` and the path $p = 2.2$, then $[\texttt{t}]_{2.2}$ =`t_2_2`.

- Let $s$ be a non-variable term of type $T$, and $p$ in `spath`$(s)$. Then

  – If $p$ is of the form $p_1.p_2$ with $p_1$ the path to a variable $x$ subterm of $s$, define: $[s]_p = [x]_{p_2}$.

For instance, consider a term `s= node(sum,leaf(3),x)`, with `x` defined by **`var`** `tree(2):x;`, and a path $p = 2.3.2.2$. Then, $[\texttt{s}]_{2.3.2.2}$=`x_2_2`. Observe that here 2.3 indicates first that `s` is a `node` (value `2`), and then points to its third argument (value `3`), which corresponds to variable `x`.

  – If the subterm of $s$ at $p$ is a standard value $v$, then: $[s]_p = v$.
    For instance, given `s= node(sum,leaf(3),t)`, and $p = 2.2.1.1$, $[\texttt{s}]_{2.2.1.1}$=`3`.

  – If $p$ does not contain the path of any variable, and the subterm of $t$ at $p$ is a term rooted by constructor $c$, then: $[s]_p = i$, with $i$ the textual position of $c$ in the definition of its union type. For instance, given `s= node(sum,leaf(3),t)`, and $p = 2.2$, $[\texttt{s}]_{2.2}$=`1`, indicating that the second argument of `s` corresponds to `leaf` (value 1 because it is the first constructor in the definition of `tree`.

Then, for any expressions $e_1, e_2$ of the same union type $T$:

- Let $S$ be $\texttt{spath}(e_1) \cap \texttt{spath}(e_2)$.
- If $S = \emptyset$, then $(e_1 = e_2)^{\mathcal{T}^+} = \texttt{false}$.
- If $S \neq \emptyset$, then:

$$(e_1 = e_2)^{\mathcal{T}^+} = \bigwedge_{q \in S} [e_1]_q = [e_2]_q$$

The idea is that the paths $q \in \texttt{spath}(e_1) \cap \texttt{spath}(e_2)$ represent the possible subterms that can be solutions of $e_1 = e_2$. Observe that the definition indicates implicitly that $\texttt{spath}(e_1) \cap \texttt{spath}(e_2) = \emptyset$ implies $(e_1 = e_2)^{\mathcal{T}^+} = \texttt{false}$. An example may help to understand this transformation. Consider again the same type for arithmetic operations, this time with a tree of level 3.

```
enum op = {sum , minus , prod , div};
enum tree = { leaf(int),  node(op, tree, tree) };
var tree(2):t;
var tree(3):s;
constraint node(sum, node(minus, leaf(6), leaf(8)), t) = s;
```

Thus, here $e_1$ = `node(sum, node(minus, leaf(6), leaf(8)), t)`, and $e_2$ = `s`. First, the set of paths are obtained:

$$\begin{aligned}
\texttt{spath}(t) = \{ &\ \varepsilon, 1.1, 2.1, 2.2, 2.3, 2.2.1.1, 2.3.1.1 \} \\
\texttt{spath}(s) = \{ &\ \varepsilon, 1.1, 2.1, 2.2, 2.3, 2.2.1.1, 2.3.1.1, 2.2.2.1, 2.2.2.2, 2.2.2.3, \\
&\ 2.2.2.2.1.1, 2.2.2.3.1.12.3.2.1, 2.3.2.2, 2.3.2.3, 2.3.2.2.1.1, \\
&\ 2.3.2.3.1.1 \} \\
\texttt{spath(node(sum, node(minus, leaf(4), leaf(3)), t))} &= \\
= \{ &\ \varepsilon, 2.1, 2.2, 2.3, 2.2.2.1, 2.2.2.2, 2.2.2.2.1.1, 2.2.2.3, 2.2.2.3.1.1, \\
&\ 2.3.1.1, 2.3.2.1, 2.3.2.2, 2.3.2.3, 2.3.2.2.1.1, 2.3.2.3.1.1 \}
\end{aligned}$$

Thus

$$\begin{aligned}
\texttt{spath}(e_1) \cap \texttt{spath}(e_2) = \{ &\ \varepsilon, 2.1, 2.2, 2.2.2.1, 2.2.2.2, 2.2.2.3, 2.2.2.2.1.1, \\
&\ 2.2.2.3.1.1, 2.3, 2.3.1.1, 2.3.2.1, 2.3.2.2, \\
&\ 2.3.2.3, 2.3.2.2.1.1, 2.3.2.3.1.1 \}
\end{aligned}$$

The transformation contains one equality for each of the elements in this intersection, as displayed in the following table

| path | equality constraint |
|---:|:---|
| $\varepsilon$ | $2 = s$ |
| 2.1 | $1 = s\_2\_1$ |
| 2.2 | $2 = s\_2\_2$ |
| 2.2.2.1 | $2 = s\_2\_2\_2\_1$ |
| 2.2.2.2 | $1 = s\_2\_2\_2\_2$ |
| 2.2.2.3 | $1 = s\_2\_2\_2\_3$ |
| 2.2.2.2.1.1 | $6 = s\_2\_2\_2\_2\_1\_1$ |
| 2.2.2.3.1.1 | $8 = s\_2\_2\_2\_3\_1\_1$ |
| 2.3 | $t = s\_2\_3$ |
| 2.3.1.1 | $t\_1\_1 = s\_2\_3\_1\_1$ |
| 2.3.2.1 | $t\_2\_1 = s\_2\_3\_2\_1$ |
| 2.3.2.2 | $t\_2\_2 = s\_2\_3\_2\_2$ |
| 2.3.2.3 | $t\_2\_3 = s\_2\_3\_2\_3$ |
| 2.3.2.2.1.1 | $t\_2\_2\_1\_1 = s\_2\_3\_2\_2\_1\_1$ |
| 2.3.2.3.1.1 | $t\_2\_3\_1\_1 = s\_2\_3\_2\_3\_1\_1$ |

For instance, the equality constraint $2 = s\_2\_2\_2\_1$ corresponds using our alternative notation to `minus` $= s_{\texttt{node(\_,node(*,\_,\_),\_)}}$. Solving the constraint obtained as conjunction of all the equities that appear in the second column of the table ( together with the constraints associated to the variable definitions) yields values representing terms of the form:

```
t= leaf(a)
s= node(sum, node(minus, leaf(6),leaf(8)), leaf(a) )
```

with `a` an integer, and also solutions of the form:

```
t= node(o,leaf(a), leaf(b))
s= node(sum, node(minus, leaf(6),leaf(8)),
             node(o,leaf(a),leaf(b))
```

with `o` any `op` value, `a`, and `b` integers.

*3.5. Case statements*

We introduce *case* statements to facilitate the decomposition of the union type expressions. Each *case branch* includes a pattern representing a possible form of the case expression. These patterns can include new variables representing arbitrary subterms. An interesting property of these variables is that they do not need to be declared, and that they do not generate new variables in the transformed code. Instead they are used as references to the corresponding subterms. Case statements have the following syntax:

```
caseStatement  ⟶  case exp of branch* endcase;
       branch  ⟶  pat --> boolExp;
```

where *pat* is a cterm with the same type as *exp* and *boolExp* is a Boolean expression. A particularity of *pat* is that it only allows new variables not defined in the context. These variables are called from now on *pattern variables* and occur also in *boolExp*. Moreover, we assume also that *pat* is linear, meaning that it contains no repeated variable.

The use of *case* statements is useful for navigating the structure of the union type expressions. The transformation detects whether a branch condition cannot hold, not expanding further that case. This approach is useful for “cutting” recursive predicate calls when the recursion is based on the structure of the terms.

The transformation of a case expression of the form **`case`** *exp* **`of`** $c_1$ `->` $b_1$; ...; $c_n$ `->` $b_n$; **`end`**; is defined as $\bigwedge_{i=1}^{n} (\text{exp}, c_i\text{->}b_i)^{\mathcal{T}^+}$, where the auxiliary $(\texttt{exp}, \texttt{pat -> boolExp})^{\mathcal{T}^+}$ transforms a particular branch following the next steps:

1. First obtain the MiniZinc expression $e_1$ defined as $e_1 = (pat = exp)^{\mathcal{T}^+}$. The transformation of $(exp = pat)^{\mathcal{T}^+}$ is done considering all the variables in `pat` as variables of level 0 and with the suitable type for the position they have in `pat`.
2. Let $e_2$ be the result of removing (replacing by `true`) from $e_1$ all the constraint equalities of the form $x = e$ with $x$ a pattern variable of `pat`. Let $\theta$ be a substitution such that for each $x = e$ in $e_1$ with $x$ a pattern variable $\theta(x) = e$.
   If the domain of $\theta$ does not include all the pattern variables, then $(\texttt{exp}, \texttt{pat -> boolExp})^{\mathcal{T}^+}$ = `true` and the transformation of this branch is finished.
3. Else $(\texttt{exp}, \texttt{pat -> boolExp})^{\mathcal{T}^+} = e_2$ `->` `boolExp`$\theta$

*3.6. Predicate and Function calls*

With the introduction of recursive types, the use of recursive predicates/-functions becomes a necessity. Our setting proposes the acceptance of recursive predicates and functions. We concentrate here on the transformation of predicates, but analogous ideas can be employed for functions. The transformation is in fact the standard idea of replacing the calls by the predicate bodies. The process is repeated until the recursion ends. In order to avoid infinite recursion, the system checks that in the recursive call at least one of the union parameters has reduced its level. Otherwise the transformation finishes returning an error.

Consider a (possibly recursive) predicate $p$ defined as

**`predicate`** `p (`$t_1$`:` $a_1$`, ...` $t_n$`:` $a_n$`) = exp;`

where $t_i$ and $a_i$ stand for the type and name of the arguments respectively, with $i = 1 \ldots n$. Then each predicate call $p(v_1, \ldots, v_n)$ is transformed by:

1. Defining a substitution $\theta = \{a_1 \mapsto p_1, \ldots, a_n \mapsto p_n\}$.
2. Replacing the call by $(exp\theta)^{\mathcal{T}^+}$.

The end of the recursive calls is obtained naturally when the body of the predicate is defined by a *case* statement, as the following example extracted from Figure 7 shows:

```
enum op = {sum , minus , prod , div};
enum tree =  {leaf(int),  node(op, tree, tree)};
var tree(2):t;

predicate positive(var tree:s) =
   case s of
        leaf(x)       --> x>0;
        node(o,s1,s2) --> positive(s1) /\ positive(s2);
   endcase;

constraint positive(t);
```

In this example the transformation replaces the call `positive(t)` by

```
   case t of
        leaf(x)       --> x>0;
        node(o,s1,s2) --> positive(s1) /\ positive(s2);
   endcase;
```

The transformation of the *case* statement yields (the details are omitted for the sake of simplicity, since they are explained in detail in the following recursive call):

$$(t = \texttt{leaf} \rightarrow t_{\texttt{leaf}(*)} > 0) \wedge$$
$$(t = \text{node} \rightarrow \texttt{positive}(t_{\text{node}(\_,*,\_)}) \wedge \texttt{positive}(t_{\text{node}(\_,\_,*)}))$$

Then, the call $\texttt{positive}(t_{\text{node}(\_,*,\_)})$ is transformed as:

```
   case t_node(_,*,_) of
        leaf(x)       --> x>0;
        node(o,s1,s2) --> positive(s1) /\ positive(s2);
```

```
    endcase;
```

Observe that at this point $t_{\texttt{node(\_,*,\_)}}$ is considered a variable of type `tree` with level 1, which means that it only admits terms of the form `leaf(x)` that is, $\texttt{path}(t_{\texttt{node(\_,*,\_)}}) = \{\varepsilon, 1.1\}$.

In order to transform the first branch, first the associated equality constraint `leaf(x) =` $t_{\texttt{node(\_,*,\_)}}$ must be transformed.

Since $\texttt{path}(\texttt{leaf(x)}) = \{\varepsilon, 1.1\}$ the two sides share the same set of paths, and the transformed equality is

$$\texttt{leaf} = t_{\texttt{node(\_,*,\_)}} \ \wedge \ x = t_{\texttt{node(\_,leaf(*),\_)}}$$

Extracting the substitution associated to the pattern variables, $\theta = \{x \mapsto t_{\texttt{node(\_,leaf(*),\_)}}\}$, and applying this substitution to the right-hand side of the branch yields the transformed code associated to this branch:

$$t_{\texttt{node(\_,*,\_)}} = \texttt{leaf} \rightarrow t_{\texttt{node(\_,leaf(*),\_)}} > 0$$

The transformation for the second branch is analogous: first obtain the transformation of `node(o,s1,s2) =` $t_{\texttt{node(\_,*,\_)}}$, as explained in subsection 3.4. Considering that $\texttt{path}(\texttt{node(o,s1,s2)}) = \{\varepsilon, 2.1, 2.2, 2.3\}$, and that we have seen above that $\texttt{path}(t_{\texttt{node(\_,*,\_)}}) = \{\varepsilon, 1.1\}$, the only common path is $\varepsilon$, that is `node(o,s1,s2) =` $t_{\texttt{node(\_,*,\_)}}$ is transformed into `node =` $t_{\texttt{node(\_,*,\_)}}$.

This transformation does not include values for the pattern variables `o`, `s1`, `s2`, and therefore this branch is transformed simply into the constant `true`. The transformation of the call `positive(`$t_{\texttt{node(\_,\_,*)}}$`)` is analogous and thus no new recursive calls are produced. Thus, the initial constraint is transformed into:

**constraint**

$$\begin{array}{lll}
(t = \texttt{leaf} & \rightarrow & t_{\texttt{leaf(*)}} > 0) \wedge \\
(t = \texttt{node} & \rightarrow & ((t_{\texttt{node(\_,*,\_)}} = \texttt{leaf} \rightarrow t_{\texttt{node(\_,leaf(*),\_)}} > 0 \wedge \texttt{true}) \wedge \\
 & & (t_{\texttt{node(\_,\_,*)}} = \texttt{leaf} \rightarrow t_{\texttt{node(\_,\_,leaf(*))}} > 0 \wedge \texttt{true})));
\end{array}$$

*3.7. Predefined function* show

In this subsection $[c]_s$ denotes the string representation of a constructor $c$. That is $[\texttt{tree}]_s =$ "`tree`".

The transformation of `show(t)`, $(\texttt{show(t)})^{\mathcal{T}^+}$, depends on the type and form of $t$:

1. If the type of $t$ is standard, then it is kept unaltered in the transformed program $(\texttt{show(t)})^{\mathcal{T}^+} = \texttt{show(t)}$.
2. If $t$ is a c-rooted term of a union type of the form $c(t_1, \ldots, t_n)$ then:

$$
\begin{aligned}
(\texttt{show(t)})^{\mathcal{T}^+} = \; [c]_s \;{+}{+}\; & \texttt{"("} \;{+}{+} \\
& (\texttt{show(}t_1\texttt{)})^{\mathcal{T}^+} \;{+}{+}\; \texttt{","} \;{+}{+} \\
& \ldots \\
& (\texttt{show(}t_n\texttt{)})^{\mathcal{T}^+} \;{+}{+}\; \texttt{")"}
\end{aligned}
$$

3. If $t$ is a variable $x$ defined as **var** `T(l):x;`, with

$$T \equiv c_1\ t_1^1 \ldots t_{m_1}^1 \mid \cdots \mid c_n\ t_1^n \ldots t_{m_n}^n$$

Then, $(\texttt{show(x)})^{\mathcal{T}^+} = \texttt{sVar(T,x,l,1)}$, with `sVar(T,x,l,i)` defined as follows:

```
If $i > n$ then return "" (the empty string)
If $l = 0$ and $m_i > 0$ then return sVar(T,x,l,i+1)
Else
    If $m_i = 0$ return the following MiniZinc statement:
     **if** (fix(x=i))
     **then** $[c_i]_s$
     **else** sVar(T,x,l,i+1)
     **endif**
    Else return
     **if** (fix(x=i))
     **then** $[c_i]_s$ ++"(" ++
        sVar($t_1^i, x_{i1}, l-1, 1$) ++"," ++ ... ++
        sVar($t_{m_i}^i, x_{im_i}, l-1, 1$) ++")"
     **else** sVar(T,x,l,i+1)
     **endif**
```

The auxiliary transformation *sVar* is defined recursively, traversing all the possible constructors in the type of the variable. The first line, corresponds to the basic case, where the empty string is returned if we have already considered all the possible values of $i$. The next condition “If $l = 0$ and $m_i > 0$ then” corresponds to the case of variable with level 0 and a constructor with a positive number of arguments. The constraints associated to the variable ensure that this case cannot occur, and consequently in this case we proceed considering the next argument. Otherwise a MiniZinc **if** statement is generated ensuring that if the variable takes the value $i$ the string corresponding to the constructor and its associated arguments are displayed.

For instance, given a variable declaration `var tree(2):t`, the MiniZinc$^+$ code `output([show(t)]);` is transformed into the MiniZinc code which is partially shown here:

```
output([show(
if (fix(t) = 1)
then "leaf" ++ "(" ++ show(t_1_1) ++ ")"
else if ((fix(t) = 2))
     then "node" ++ "(" ++
             (if (fix(t_2_1) = 1) then "sum"
              ...
              else if (fix(t_2_1) = 4) then "div"
              else ""
              ...endif endif )
              ++ "," ++
              (if (fix(t_2_2) = 1)
              then "leaf" ++ "(" ++ show(t_2_2_1_1) ++ ")"
              ...
     else ""
     endif
endif)]);
```

The transformation of the program of Figure 7 following the ideas explained in this section (not included here for the sake of space) displays as first solution:

```
node(minus,
     node(minus,
          node(sum, leaf( 3), leaf( 32)),
          leaf(14)),
     leaf( 14))
```

which corresponds to the arithmetic expression (3+32)-14-14 = 7, thus solving the problem established in our running example.

*3.8. Experimental Results*

The implementation of the prototype is available at `https://github.com/RafaelCaballero/MiniZincU`. This subsection discusses the efficiency of three different examples using union datatypes.

The code of the first example can be seen in Figure 9. The model includes two binary trees `t` and `s`, whose nodes contain integers in the range `-100..100`. The level (that, the upper bound of the height) of the two trees is determined by the parameter `N`. In the constraint section we require that both trees contain at least all the integers $i$ such that $0 \leq i \leq N$. This represents $N + 1$ different nodes, which can be stored in a binary tree of height $N \geq 2$ (a binary tree of level $N$ can contain up to $2^N - 1$ nodes, and $2^N - 1 \geq N + 1$ for every $N \geq 2$). Moreover, the maximum value contained in $s$ must be greater than the maximum value in $t$, which implies that $s$ contains at least an additional value $y > N$. Finally, the goal is to minimize the value $y$, which in this case corresponds to $y = N + 1$. The constraints rely on the predicates `contains` and `maxVal`. The table in the same figure shows the space and time growth with respect to the level `N` displayed in the first column. The description of the rest of the columns is the following:

- *Trans. Time*: Time in seconds required by the transformation from MiniZinc$^+$ into MiniZinc.
- *Size:* Size in Kilobytes of the transformed code (that is the MiniZinc code obtained after the transformation).
- *Num. Vars:* Number of variables generated in MiniZinc by each MiniZinc$^+$ union variable. For instance, for level $N = 6$, the MiniZinc code contains $253 \times 2$ variables (253 corresponding to $t$ and 253 to $s$).
- *Sol. Time*: Time required to obtain the first solution in MiniZinc.

```
int:N=9;
set of int: Int=-100..100;
enum tree = { leaf(Int), node(Int, tree, tree)};
var tree(N):t;  var tree(N):s;
var Int:x;      var Int:y;

predicate contains(var tree:t, var Int:r) =
case t of
leaf(o) --> r=o;
node(o,t1,t2) --> r=o \/contains(t1,r) \/contains(t2,r);
endcase;
predicate maxVal(var tree:t, var Int:r) =
case t of
leaf(o) --> r=o;
node(o,t1,t2) -->  let {var Int:r1} in (maxVal(t1,r1) /\
                   let {var Int:r2} in (maxVal(t2,r2) /\
                    (r=max([r1,r2,o]))));
endcase;
constraint forall(i in 0..N)
                (contains(t,i) /\ contains(s,i));
constraint maxVal(t,x) /\ maxVal(s,y) /\ x<y;
solve minimize y;
output([show(t), show(s)]);
```

| Level(N) | Trans. Time | Size | Num. Vars. | Sol. Time |
|---|---|---|---|---|
| 2 | 750 ms. | 5 Kb | 13 × 2 | 36 ms. |
| 3 | 754 ms. | 14 Kb | 29 × 2 | 55 ms. |
| 4 | 816 ms. | 36 Kb | 61 × 2 | 127 ms. |
| 5 | 989 ms. | 89 Kb | 125 × 2 | 316 ms. |
| 6 | 1312 ms. | 218 Kb | 253 × 2 | 1249 ms. |
| 7 | 2205 ms. | 523 Kb | 509 × 2 | 2851 ms. |
| 8 | 6110 ms. | 1220 Kb | 1021 × 2 | 6524 ms. |
| 9 | 26905 ms. | 2846 Kb | 2045 × 2 | 12891 ms. |

Figure 9: Containment and max. value of trees in MiniZinc$^+$

The table shows the exponential growth of time and space with respect to $N$. This is not surprising because the number of possible nodes in the trees grows also exponentially with respect to $N$. In particular, the table shows that number of new variables is proportional to the number of possible nodes in the tree (with an approximate factor of 4, for instance with $N = 8$ there are $1021 \sim 4 \times (2^8 - 1)$ variables). The file size (which determines the compilation time), is influenced by the number of variables and their related constraints. In the case of the constraints, it is important to notice that they include not only those generated by the transformation of the variables, but especially the constraints obtained by unfolding the recursive predicate calls. An additional overhead is caused by the **`output`** statement: an expression like **`show`**`(t)`, with `t` of level $N = 9$, is transformed into an expression that needs to check all the possible tree configurations with height less than 10, and this requires approximately 11Kb of standard MiniZinc code.

Therefore, models using non-linear structures like trees can lead to a 'combinatorial explosion' which affects the size, time of compilation, and often execution time. It is worth noticing that in many cases the exponential growth is difficult to overcome. For instance, if we wish to represent binary trees of height less than or equal to $N$ in standard MiniZinc, it is natural that we need a number of variables proportional to $2^N - 1$ in order to store the values at the nodes, plus additional constraints to control which nodes are indeed in the tree and which are not needed in a particular solution.

Our prototype includes an optimizer that can help to improve these figures in some particular models. The idea is to detect whether the constraints imposed over the data structure force the values of some variables to take only one possible value. In such cases the optimizer can substitute the variable by the constant, thus reducing the number of variables and constraints generated in the transformed model.

An extreme case in this sense is the model of Figure 10, which represents a fully complete tree with integer nodes, that is, a tree where the path to every leaf contain exactly $N$ nodes. Observe that this does *not* determine a unique tree, the nodes can still contain any possible integer. What is determined is the structure of the tree. The figures in the corresponding table show a much more scalable problem, with the number of variables reduced to the minimum necessary to contain the tree nodes, and the size of the transformed model much smaller that in Figure 9 for trees of the same size.

The behavior of the optimizer can be better understood considering a simple example with $N = 2$. With this level, the transformation initially

```
int:N=9;

enum tree = { leaf(int), node(int, tree, tree)};

var tree(N):t;

predicate compl(var tree:t, var int:l) =
case t of
leaf(x) --> l=1;
node(o,t1,t2) --> l>0 /\
                  compl(t1,l-1) /\ compl(t2,l-1);
endcase;

constraint compl(t,N);

solve satisfy;

output([show(t)]);
```

| Level(N) | Trans. Time | Size | Num. Vars. | Sol. Time |
|---|---|---|---|---|
| 2 | 553 ms. | <1 Kb | 3 | 20 ms. |
| 3 | 552 ms. | 1 Kb | 7 | 21 ms. |
| 4 | 557 ms. | 3 Kb | 15 | 24 ms. |
| 5 | 642 ms. | 7 Kb | 31 | 23 ms. |
| 6 | 735 ms. | 15 Kb | 63 | 37 ms. |
| 7 | 1247 ms. | 32 Kb | 127 | 67 ms. |
| 8 | 2218 ms. | 67 Kb | 255 | 311 ms. |
| 9 | 4639 ms. | 138 Kb | 511 | 762 ms. |

Figure 10: A fully complete tree in MiniZinc$^+$

generates among other the following constraints for the model of Figure 10:

```
....
constraint (t = 2) /\ ((t_2_2 = 1) /\ (t_2_3 = 1));
constraint t_2_3 != 2 ->
          ((t_2_3_2_1 = 0) /\
           (t_2_3_2_2 = 1) /\
           (t_2_3_2_3 = 1));
...
```

From the first constraint the optimizer extracts the substitution $\{t \mapsto 2, t\_2\_2 \mapsto 1, t\_2\_3 \mapsto 1\}$. After applying the substitution the variables can be removed, and the constraints are now:

```
....
constraint (2 = 2) /\ ((1 = 1) /\ (1 = 1));
constraint 1 != 2 ->
          ((t_2_3_2_1 = 0) /\
           (t_2_3_2_2 = 1) /\
           (t_2_3_2_3 = 1));
...
```

The first constraint is equivalent to `true` and is readily removed. In the second constraint the implication condition is also `true`, and this yields a new substitution $\{(t\_2\_3\_2\_1 \mapsto 0, t\_2\_3\_2\_2 \mapsto 1, t\_2\_3\_2\_3 \mapsto 1\}$. Applying the same technique until no more substitutions are found, the optimizer reduces the total number of variables from 13 to 3 in this particular case, and all the model constraints are removed. Although the optimizer rarely has such huge impact on the transformation performance, it is still useful in real cases where part of the structure is determined by the constraints (for instance if a constraint specifies that the tree cannot be a single leaf).

The last example shows a model defining a linear data structure, in this case a stack of integer numbers. In particular, the MiniZinc$^+$ code of Figure 11 defines two stacks `a` and `b`, containing `la` and `lb` elements respectively.The last constraint indicates that $0 < la < lb$, that is both stacks are non-empty and `b` contains more elements than `a`. The table shows that this model scales very well when the size of the stack is increased. In particular notice that the

```
int:N=9 ;
enum stack  = { empty, s(int,stack) };
var stack(N):a;
var stack(N):b;
var int:la;
var int:lb;

predicate length(var stack:s, var int: x) =
case s of
 empty --> x=0;
 s(n,s2) --> let {var int:x2} in
                length(s2,x2) /\ x=x2+1;
endcase;

constraint length(a,la);
constraint length(b,lb);
constraint la>0 /\ la<lb;

solve satisfy;

output(["a: ",show(a),"\n",
        "b: ",show(b),"\n"
       ]);
```

| Level(N) | Trans. Time | Size | Num. Vars. | Sol. Time |
|---|---|---|---|---|
| 2 | 591 ms. | 2 Kb | $5 \times 2$ | 21 ms. |
| 3 | 585 ms. | 3 Kb | $7 \times 2$ | 24 ms. |
| 4 | 598 ms. | 5 Kb | $9 \times 2$ | 25 ms. |
| 5 | 594 ms. | 6 Kb | $11 \times 2$ | 26 ms. |
| 6 | 672 ms. | 9 Kb | $13 \times 2$ | 26 ms. |
| 7 | 634 ms. | 11 Kb | $15 \times 2$ | 27 ms. |
| 8 | 672 ms. | 14 Kb | $17 \times 2$ | 48 ms. |
| 9 | 735 ms. | 17 Kb | $19 \times 2$ | 31 ms. |

Figure 11: Stacks in MiniZinc$^+$

number of generated variables is linear with respect to $N$ (in fact is $2N - 1$ for a stack of level $N$).

In any case, it must be noticed that a source-to-source transformation is not very suitable for obtaining a good performance. Our goal when implementing this prototype was to check whether the theoretical ideas gave rise to a reasonable framework for defining unions, and explore the limitations of the proposal. In order to achieve a better performance these extensions should be integrated as part of the MiniZinc system.

## 4. Related Work

The closest related work to MiniZinc$^{\star}$ is a special case of extended types also implemented in MiniZinc called *option types* [16]. Option types add an additional value $\top$ to a type which acts as a identity element where possible. While it might appear that examples we use, for example the integers with NULL `intN` could be modelled with option types, the desired behaviour of NULL differs from $\top$. Since we can program the behaviour of extended types we can implement the desired behaviour.

The closest work to MiniZinc$^{+}$ is work on type reduction in Zinc [15]. Zinc uses data indepedent type reduction to rewrite tuple types and record types to simpler language constructs. The type reduction in Zinc is done independent of the data, unlike the approach we use here for MiniZinc$^{+}$. Type reduction of union types is not supported in Zinc, and indeed recursive discriminated union types are not allowed in the language.

Another related work is [18], which enables the user to define open domains, i.e., domains where values do not need to be explicitly and exhaustively listed, but their elements can be acquired along the computation, when needed. This approach could be useful in a future implementation of type extensions as part of the system MiniZinc.

## 5. Conclusions and Future Work

We have presented two extensions of the MiniZinc type system that allow the representation of many constraint satisfaction problems in a more natural way.

The first proposal allows extending predefined types with new constants. Some examples are models representing circuits including undefined entries (representing for instance failing connections), database problems including *null* values, problems that are better modelled using non-classical logics that do not restrict the number of truth values to only *true* and *false* [5], or scheduling problems with optional tasks (although for these scheduling problems there are approaches [17] which allow using *time-interval variables* and extend Constraint-Based scheduling to efficiently propagate on).

The second proposal allows to define union datatypes in the models, which is specially useful in problems that can be naturally expressed using recursive datatypes. In practice the level of recursion must be initially bound, but the new types still are useful for modelling interesting problems which are naturally represented by data structures such as trees. The main limitation of the proposal is the exponential number of variables that can be generated during the transformation into MiniZinc. An interesting line of future research is to incorporate the union datatypes into the language in a way that permits to create the variables dynamically, thus limiting the number of variables to those necessary to represent the particular solution.

It is worth noticing that, although presented separately, both transformations can coexist, or even be combined in the same datatype: although extending a union type makes little sense, it can be interesting for instance to define trees whose nodes contain extended integers.

Clearly the modeller could directly use MiniZinc rather than MiniZinc$^\star$, MiniZinc$^+$, or their combination to model their problem (since both MiniZinc$^\star$ and MiniZinc$^+$ are implemented by translation), but the direct model is much less concise and much harder to get right since extended types can interact in complex ways. Our experience in creating large models using extended types by hand was that it was very difficult, motivating our need for this work.

We present a model transformation that converts the models in the new type systems into standard MiniZinc models. Thus, all the facilities included in MiniZinc such as intensional lists, local definitions, sets, or predicates are available in the new setting.

We establish the correctness of the proposed transformation at the semantic level. This implies formalizing a suitable semantics for MiniZinc and the proposed extensions, which is interesting by itself.

Regarding efficiency, we think that the implementation is acceptable for small to medium-size models, although it can become a problem in the case of big models (or complex models that require big data structures, in the case of union types). In order to improve the performance, another approach would be to implement a new underlying solver that takes into account these features. For instance, the solver will generate extra variables only when they are required. As mentioned in the introduction, we have considered instead a source-to-source transformation for the sake of clarity in the explanation, the possibility to prove the soundness of the approach, and because transcompiling to MiniZinc means the possibility of using all the solvers that accept FlatZinc.

The main line of future research would be to integrate these features in the standard MiniZinc system.

## Appendix A. Theoretical results

In this section we present the theoretical results that support our proposal. The idea is to prove that both the MiniZinc$^\star$ (respectively MiniZinc$^+$) model and its transformation represent the same set of solutions. The solutions are represented by well-typed substitutions:

**Definition 1.** *Let $\mathcal{M}$ be a MiniZinc$^\star$ model, $\Gamma$ its associated type context, and $\theta$ a substitution. We say that $\theta$ is a* well-typed *substitution for $\mathcal{M}$ iff*

- *The domain of $\theta$ is the set containing all the decision variables declared in $\mathcal{M}$.*[10]
- *For all $x \in \mathtt{dom}(\theta)$, $\mathtt{type}(x) = \langle t \rangle$ iff $\mathtt{type}(x\theta) = \langle t \rangle$.*

For instance, in the type extension example of Figure 2 the following is a part of a well-typed substitution (excluding *let* variables for simplicity):

$$\theta_1 = \left\{ \begin{array}{l} n \mapsto 4, \\ x \mapsto \texttt{[true | undef | false, true]}, \\ y \mapsto \texttt{[true | undef | false, false]}, \\ c \mapsto \texttt{[false | true | undef | false, false]}, \\ s \mapsto \texttt{[false | undef | undef | true, false]}, \\ \ldots \end{array} \right\}$$

where the notation `[... | ...]` is employed to represent arrays. Analogously, in the case of the union types example of Figure 7, again omitting *let* variables, the following is a well-typed substitution.

$$\theta_2 = \left\{ \begin{array}{l} \texttt{t} \mapsto \texttt{node(subst,} \\ \qquad\qquad \texttt{node(subst,} \\ \qquad\qquad\qquad \texttt{node(add, leaf(32), leaf(3)),} \\ \qquad\qquad\qquad \texttt{leaf(14)),} \\ \qquad\qquad \texttt{leaf(14))} \\ \ldots \end{array} \right\}$$

[10] The decision variables are the variables declared either at top level, in local *let* statements, or as pattern variables. The parameter names in the declarations of user functions and predicates are *not* considered decision variables in our setting.

The concept of solution is based on the evaluation of an expression in a model with respect to a given well-typed substitution.

**Definition 2.** *Let $\mathcal{M}$ be a MiniZinc$^\star$ model, $e$ an expression occurring in $\mathcal{M}$, and $\theta$ be a well-typed substitution for $\mathcal{M}$. The* evaluation of $e$ with respect to *$\theta$, denoted by $\| e \|_\theta$, is defined distinguishing cases according to the definition of MiniZinc$^\star$ expressions (refer to non-terminal* exp *in the grammar)*

1. $\| id \|_\theta = id\theta$, id *any identifier.*
2. $\| k \|_\theta = k$, k *any constant.*
3. *Set Expressions:*
   (a) $\| \{e_1, \ldots, e_n\} \|_\theta = \mathtt{ord}(\{\| e_1 \|_\theta, \ldots, \| e_n \|_\theta\})$.
   *$\mathtt{ord}$ is defined as the function that given a set of values, eliminates the repetitions and sorts the values according to the order $\preceq$ that extends $\mathtt{ord}_t$ defined in Section 2.5 where:*

$$a \preceq b = \begin{cases} a \le b & a, b \textit{ standard} \\ \mathtt{ord}_t(a) < 0 & a \textit{ ext.}, b \textit{ std.} \\ \mathtt{ord}_t(b) > 0 & a \textit{ std.}, b \textit{ ext.} \\ \mathtt{ord}_t(a) \le \mathtt{ord}_t(b) & \textit{otherwise} \end{cases}$$

   (b) $\| e_i..e_f \|_\theta = \{\| e_i \|_\theta, \| e_i \|_\theta + 1, \ldots, \| e_f \|_\theta\}$
4. *Array Expressions:* $\| [e_1, \ldots, e_n] \|_\theta = [\| e_1 \|_\theta, \ldots, \| e_n \|_\theta]$
5. *Array Access:*
   (a) $\| a[e] \|_\theta = t_i$, *with* $\mathtt{a}$ *an array identifier with index range* $m \ldots n$, $i = \| e \|_\theta - m + 1$, $1 \le i \le n - m + 1$, *and* $\| a \|_\theta = [t_1, \ldots, t_{n-m+1}]$.
   (b) $\| e_1[e_2] \|_\theta = t_i$, *with* $e_1$ *not an array identifier,* $\| e_1 \|_\theta = [t_1, \ldots, t_n]$, *and* $i = \| e_2 \|_\theta$, $1 \le i \le n$.
6. *Set/list comprehensions of the form $lc = \langle e \mid g_1, \ldots, g_m$ where $c\rangle$, where:*
   (a) *$\langle , \rangle$ represents either {,} or [,].*
   (b) *$g_j$ is of the form $id_j$ in arrayexp or $id_j$ in setexp.*
   (c) *In particular suppose that $g_1 \equiv id$ in $e'$. Let $\| e' \|_\sigma$ be $\langle\, e_1, \ldots, e_n \,\rangle$ and define*

$$\sigma_1 = \sigma \uplus \{id \mapsto e_1\}, \ldots, \sigma_n = \sigma \uplus \{id \mapsto e_n\}$$

   *Moreover, in the definition we use the following notation:*

   - *$\diamond$ represents the array concatenation or set union depending on what $\langle , \rangle$ is representing.*

- $\mathcal{C}(e, c)$ *being* $\langle e \rangle$ *if* $c$ *holds and* $\langle\rangle$ *in other case.*

*Then,* $\| \, lc \, \|_\theta$ *is defined recursively as:*

(a) *If* $m = 1$*, then* $lc$ *contains only one generator* $g$*, which must be of the form* $id$ *in* $e'$*. Then:*

$$\| \, \langle e \; | g \textit{ where } c \rangle \, \|_\sigma = \\ \mathcal{C}(\| \, e \, \|_{\sigma_1}, \| \, c \, \|_{\sigma_1}) \; \diamond \ldots \diamond \mathcal{C}(\| \, e \, \|_{\sigma_n}, \| \, c \, \|_{\sigma_n})$$

(b) *If* $m > 1$ *then* $lc$ *contains more than one generator. Analogously to the previous item, suppose that the first generator is* $g_1$*. Then:*
$\| \, \langle e \; | g_1, \ldots, g_m \textit{ where } c \rangle \, \|_\sigma =$
$\| \, \langle e \; | g_2 \ldots, g_m \textit{ where } c \rangle \, \|_{\sigma_1} \; \diamond \ldots \diamond$
$\| \, \langle e \; | g_2 \ldots, g_m \textit{ where } c \rangle \, \|_{\sigma_n}$

7. $\|$ `if` $c$ `then` $e_1$ `else` $e_2$ `endif` $\|_\theta =$
   - $\| \, e_1 \, \|_\theta$*, if* $\| \, c \, \|_\theta = \textit{true}$*.*
   - $\| \, e_2 \, \|_\theta$*, if* $\| \, c \, \|_\theta = \textit{false}$*.*
8. $\| \, let \; \{d_1, \ldots, d_n, \; c_1, \ldots, c_n\} \; in \; e \, \|_\theta = \| \, e \, \|_\theta$*, if* $(\| \, c_1 \, \|_\theta = \textit{true} \wedge \ldots \wedge \| \, c_n \, \|_\theta = \textit{true})$
9. $\| \, sv([e_1, \ldots, e_n]) \, \|_\theta = st(t_1) \wedge \cdots \wedge st(t_n)$ *with* $\Gamma \vdash \| \, e_1 \, \|_\theta :: t_1, \Gamma \vdash \| \, e_n \, \|_\theta :: t_n$
10. $\| \, e_1 = e_2 \, \|_\theta =$ `true` *if* $\| \, e_1 \, \|_\theta$ *and* $\| \, e_2 \, \|_\theta$ *are the same constant,* `false` *otherwise.*
11. $\| \, p(e_1, \ldots, e_n) \, \|_\theta = p(\| \, e_1 \, \|_\theta, \ldots, \| \, e_n \, \|_\theta)$ *, with* $p$ MiniZinc *predefined (that* $p$ *is a relational operator or predefined arithmetic function such as* $>$,$<$,$+ \ldots$*) .*
12. $\| \, p(e_1, \ldots, e_n) \, \|_\theta$*, with* $p$ *a user-defined predicate or function. Suppose that* $p$ *is defined as* predicate p($d_1$,..., $d_n$) = e *(analogous for functions). By construction each* $d_i$ *has an associated decision variable/parameter identifier* $x_i$*. Then,*

$$\| \, p(e_1, \ldots, e_n) \, \|_\theta = \| \, e \, \|_{\theta \uplus \{x_1 \mapsto \| \, e_1 \, \|_\theta, \ldots, x_n \mapsto \| \, e_n \, \|_\theta\}}$$

13. Forall*,* exists *constructions:*
    *Let* $\| \, a \, \|_\theta$ *be* $[v_1, \ldots, v_n]$*, then:*
    - $\|$ `forall`$(a) \, \|_\theta = v_1 \wedge \cdots \wedge v_n$
    - $\|$ `exists`$(a) \, \|_\theta = v_1 \vee \cdots \vee v_n$

14. Case *statements:*
    $\|$ case *exp* of $c_1$ -> $b_1$*;* $\ldots$*;* $c_n$ -> $b_n$*;* end*;* $\|_\theta$ =
    $(\| exp = c_1 \|_\theta \rightarrow \| b_1 \|_\theta) \wedge \ldots \wedge (\| exp = c_n \|_\theta \rightarrow \| b_n \|_\theta)$

Thus, the overall idea is simply to evaluate the expressions after replacing the variables by their values. Observe that the case of predicates/functions, we assume that $\theta$ contain values for the local variables and pattern variables that occur in the evaluation of the expressions of the model, and that the variables associated to each call can be clearly identified.

Now we can define the concept of solution.

**Definition 3.** *Let $\mathcal{M} = T; U; D; A; C; S$ be a* MiniZinc$^\star$ *or a* MiniZinc$^+$ *model, where $T$ is a sequence of type extensions and type union declarations, $D$ the sequence of variable declarations, $A$ the sequence of assignments, $P$ and $F$ sequences of predicate and function declarations, $C$ the sequence of constraints, and $S$ the* solve *statement. Let $\theta$ be a well-typed substitution for $\mathcal{M}$. Then, we say that $\theta$ is a solution of $\mathcal{M}$ if:*

1. *For every assignment $a$ in $A$, $\| a \|_\theta = true$.*
2. *For every* constraint *$c$ in $C$, $\| c \|_\theta = true$.*
3. *If $S$ is of the form* maximize f *(respectively* minimize f*) then there is no well-typed substitution $\sigma$ for $\mathcal{M}$ verifying 1) and 2) and such that $f\sigma > f\theta$ (respectively $f\sigma < f\theta$)*

For instance the constraint included in the model of Figure 2:

```
constraint c[1]=false /\ s[n+1]=c[n+1]
```

is evaluated to `true` by substitution $\theta_1$, because

1. $\theta_1(c[1])$ =`false`
2. $\theta_1(s[5]) = \theta(c[5])$ =`false`,
3. Both `c[1]=false` and `s[n+1]=c[n+1]` are evaluated to `true`.
4. `true` $\wedge$ `true` is `true` because the redefinition of the conjunction described in Figure 2 behaves like the usual conjunction on boolean values.

In fact, it is easy to check that, adding the suitable values for local variables, substitutions $\theta_1$ and $\theta_2$ are solutions of the models presented in Figures 2 and 7, respectively.

The next definition transforms substitutions in MiniZinc$^\star$ into substitutions in MiniZinc.

**Definition 4.** *Let $\mathcal{M}$ be a* MiniZinc$^\star$ *model and $\sigma$ be a well-typed substitution of $\mathcal{M}$, then we define a substitution $\sigma^{\mathcal{T}^\star}$*

$$\begin{array}{rl}\sigma^{\mathcal{T}^\star} = & \{\tau_s(x) \mapsto \tau_s(v) \mid (x \mapsto v) \in \sigma\} \cup \\ & \{\tau_e(x) \mapsto \tau_e(v) \mid (x \mapsto v) \in \sigma,\, \Gamma \vdash x :: t,\, \tau_e(x) \neq z_t\}\end{array}$$

For instance, in the case $\theta_1$ we can obtain the substitution:

$$\theta_1{}^{\mathcal{T}^\star} = \left\{\begin{array}{l} n \mapsto 4, \\ x_s \mapsto \texttt{[true | }\underline{\texttt{false}}\texttt{ | false, true],} \\ y_s \mapsto \texttt{[true | }\underline{\texttt{false}}\texttt{ | false, false],} \\ c_s \mapsto \texttt{[false | true | }\underline{\texttt{false}}\texttt{ | false, false],} \\ s_s \mapsto \texttt{[false | }\underline{\texttt{false}}\texttt{ | }\underline{\texttt{false}}\texttt{ | true, false],} \\ \ldots \end{array}\right\}$$
$$\cup$$
$$\left\{\begin{array}{l} x_e \mapsto \texttt{[0 | 1 | 0, 0],} \\ y_e \mapsto \texttt{[0 | 1 | 0, 0],} \\ c_e \mapsto \texttt{[0 | 0 | 1 | 0, 0],} \\ s_e \mapsto \texttt{[0 | 1 | 1 | 0, 0],} \\ \ldots \end{array}\right\}$$

where the $\underline{\texttt{false}}$ values of the first part of the substitution correspond to the translation of `undef` as standard value, while the 0's in the second part correspond to the extended part of any boolean value.

We are ready establish the theoretical result that establishes the soundness of the transformation for MiniZinc$^\star$.

**Theorem 1.** *A well-typed substitution $\theta$ is solution of a* MiniZinc$^\star$ *model $\mathcal{M}$ iff $\theta^{\mathcal{T}^\star}$ is well-typed solution of $\mathcal{M}^{\mathcal{T}^\star}$.*

**Proof Idea**

We must check that $\theta$ verifies the Definition 3 with respect to $\mathcal{M}$ iff $\theta^{\mathcal{T}^\star}$ verifies the same Definition with respect to $\mathcal{M}^{\mathcal{T}^\star}$.

For items 1 and 2, the result is a consequence of a similar auxiliary lemma applied to expressions:

*For every expression $e$ and well-typed substitution $\theta$:*

- $\| \tau_s(\| e \|_\theta) \|_{\texttt{id}} = \| \tau_s(e) \|_{\theta^{\mathcal{T}^\star}}$

- $\|\, \tau_e(\|\, e \,\|_\theta) \,\|_{\mathtt{id}} = \|\, \tau_e(e) \,\|_{\theta^{\mathcal{T}^\star}}$

*where* `id` *represents the identity substitution.* These results can be proven using structural induction on the form of $e$.

Analogously, item 3 requires a generalization of the following result: *For every pair of constants $k$, $k'$ of some type $t$ in $\mathcal{M}$ $k \leq k'$ (with the order $<$ extended to the new types) iff*

$$\tau_e(k) * (b - a + 1) + \tau_s(k) \leq \tau_e(k') * (b - a + 1) + \tau_s(k')$$

*where $a$ and $b$ are respectively the minimum and the maximum constants in the base type for $t$.*

Thus, ${\theta_1}^{\mathcal{T}^\star}$ must be a solution of the translation of the code mode of Figure 2. Analogously, we can define the transformation of a substitution in MiniZinc$^+$ to a substitution in MiniZinc.

**Definition 5.** *Let $\mathcal{M}$ be a MiniZinc$^+$ model and $\sigma$ be a well-typed substitution of $\mathcal{M}$, then we define a substitution $\sigma^{\mathcal{T}^+}$*

| | |
|---|---|
| $\sigma^{\mathcal{T}^+}(x) = x$ | *if $x$ is not obtained by the transformation of a variable of an union type* |
| $\sigma^{\mathcal{T}^+}(x_p) = [\sigma(x)]_p$ | *if $x_p$ has been obtained transforming a variable $x$ of an union type* |

For instance, since $\theta_2(t) =$ `node(subst,...,...,)`, we have that

$${\theta_2}^{\mathcal{T}^+}(\mathtt{t}_{\mathtt{node(*,\_,\_)}}) = \mathtt{subst}$$

or, using the notation employed in our implementation ${\theta_2}^{\mathcal{T}^+}$`(t`$_{\mathtt{2\_1}}$`)=2` (2 is the constant associated to `subst`). The last theoretical result is the analogous of Theorem 1 for the case of MiniZinc$^+$, and proves that ${\theta_2}^{\mathcal{T}^+}$ is a solution for the translation of the model in Figure 7:

**Theorem 2.** *A well-typed substitution $\theta$ is solution of a* MiniZinc$^+$ *model $\mathcal{M}$ iff $\theta^{\mathcal{T}^+}$ is well-typed solution of $\mathcal{M}^{\mathcal{T}^+}$.*

**Proof Idea**

As in the previous proof we must check that $\theta$ verifies the three items of Definition 3 with respect to $\mathcal{M}$ iff $\theta^{\mathcal{T}^\star}$ verifies the same items with respect to $\mathcal{M}^{\mathcal{T}^\star}$. In the case of MiniZinc$^+$ the third item is independent of the definition of union types, because in our proposal such values must be values of standard types. In order to prove the first item, assignments, we prove that there is a bijective mapping between terms in MiniZinc$^+$ and their representation in MiniZinc, and then the result holds by the definition of the substitution $\theta^{\mathcal{T}^+}$ (Def. 5). The second item refers to constraint, and since the only constraints allowed for union type terms are equality constraints we proceed checking that $\| e_1 = e_2 \|_\theta = \texttt{true}$ iff $\| e_1 = {e_2}^{\mathcal{T}^+} \|_{\theta^{\mathcal{T}^+}} = \texttt{true}$. This can be done by induction on the structure of one of the terms (for instance on the left-hand side term), distinguishing apart *case* statements and predicate/functions which deserve an auxiliary result.

| | **Type Extensions** | **Union Types** |
|---|---|---|
| **Type Extensions** | sound | - |
| **Union Types** | sound | sound (default) |

Table B.3: Soundness of possible combinations

## Appendix B. Combining Union Types and Type Extensions

So far we have presented type extensions (Section 2) and union types (Section 3) as two separated enhancements that increase the expressiveness of MiniZinc. An interesting question is whether both approaches can coexist, and how a combined model can be transformed into a standard model.

The next Subsections discuss all the possible combinations and their implementation. A summary of the results can be found in Table B.3.

### *Appendix B.1. Generalized Extended Types*

The definition of extended types, represented by the non-terminal `typeE` of the grammar presented in Figure 1, specifies that an extended type is based on an standard type plus some additional constants. Although our current implementation is limited to this case, it is possible to generalize this definition in order to allow hierarchies of extended types. We call *generalized extended types* to this new framework.

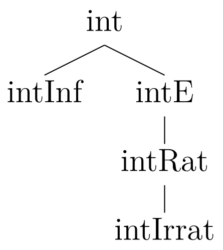
#### *Appendix B.1.1. Definition of Generalized Extended Types*

Consider for instance the generalized extended types defined in Figure B.12. Initially, two first-level extensions of the standard type **`int`** are defined: `intInf` for integers including positive and negative infinite, and `intE` for integers extended with an error value `undef`. Then, two new types are built over `intE`: `intRat` which includes a constant indicating that the number is a non-integer rational, and `intIrrat` which adds the possibility of indicating that the value corresponds to an irrational number. The addition over the hierarchy based on `intE` is also extended defining three functions redefining the operator `+`.

The relation between the different types of the example can be depicted as follows:

```
extended intInf = [negInf] ++ int ++ [posInf];
extended intE = [] ++ int ++ [undef];
extended intRat  = []++intE ++ [rationalNotInt];
extended intIrrat = [] ++ intRat ++ [irrational];

function var intE:+(var intE:a, var intE:a) =
 let{var intRat:r,
     var bool:c= sv([a,b]),
     constraint (c /\ r= (a prdf(+) b)) \/
                (not(c) /\ r= undef)
           } in r;

function var intRat:+(var intRat:a, var intRat:a)=
 let{var intRat:r,
     var bool:ca= sv([a]), var bool:cb= sv([b]),
     var bool:u= (a= undef) \/ (b= undef),
     constraint (ca /\ cb /\ r= (a prdf(+) b)) \/
                (u /\ r = undef) \/
                (ca /\ not(u) /\ not(cb) /\ r= b ) \/
                (not(ca) /\ cb /\ not(u) /\ r= a) \/
                (not(ca) /\ not(cb)) /\ r= undef)
           } in r;

function var intIrrat:+(var intIrrat:a, var intIrrat:a)=
 let{var intRat:r,
     var bool:ca= sv([a]), var bool:cb= sv([b]),
     var bool:u= (a= undef) \/ (b= undef),
     constraint (ca /\ cb /\ r= (a prdf(+) b)) \/
               (u /\ r= undef) \/
               ((ca xor cb) /\ not(u) /\ r= irrational)\/
               (not(ca) /\ not(cb)) /\ r= undef)
           } in r;
```

Figure B.12: Generalized extended MiniZinc typges



Thus, generalized extended types can be represented as tree-shaped hierarchies with the root corresponding to a base type and the rest of the nodes to extended types. This imposes the natural requirement of avoiding circular definitions.

*Appendix B.1.2. Generalizing primitives* ***sv*** *and* ***prdf***

In order to understand how the concepts of Section 2 can be applied to this more general setting we consider now the code of the three redefinitions of the operator + in Figure B.12. The first function defines the addition of two `intE` numbers, indicating that if both of them are integer values (represented by the boolean variable `c`) then the result is the usual addition obtained by applying the standard operator +, while the result is `undef` if any of the operands is `undef`. The second function defines the addition of two `intRat` values. Using a similar schema, first two local variables `ca` and `cb` are defined to check if the input parameters `a` and `b` are standard values. As in the previous function if both are standard values, then the predefined + is employed to obtain the result. However, in this generalized context it is necessary to carefully redefine the concepts of *standard* value and *predefined* operator:

- *Standard value* means now checking if an expression of some extended type $T_c$ takes a value that belongs to a type $T_p$ such that $T_p$ is the *parent* of $T_c$ in the tree of generalized extended types. Thus, the primitive **`sv`**`[a,b,...]` is applicable if all the values of the list are expressions of the same extended type and is evaluated to `true` if all of them take values belonging to the parent type.

- *Predefined operator* means now the operator defined in the parent type. For instance, the expression `(a `**`prdf`**`(+) b)` is only applicable if both `a` and `b` are variables of the same extended type and **`sv`**`([a,b])` holds. If this is the case the result corresponds to the definition of + in the parent type.

Observe also that some other concepts need to be generalized. For instance, the default value $k_{o(t)}$ associated to each standard type $t$ is now generalized to extended types in the following way: $k_{o(t)} = k_{o(t')}$ with $t'$ the base type of $t$.

Thus, in the case of function `intRat:+` of Figure B.12 we use **`sv`** to check if both values are `intE` values and keep this information in local variables `ca` and `cb` respectively. The variable `u` indicates if any of the values is `undef`. The constraint cases are the following:

- If both parameters are standard (thus `intE` values), the definition of `intE:+` is employed.

- If any of them is `undef` (`u` holds) then the result is also `undef` (this case overlaps with the previous one if both are `intE` but in this case `a` **`prdf`**`(+) b` is also `undef` and no ambiguity arises).

- If parameter `a` takes an `intE` value but the parameter `b` does not (that is, `b=rationalNotInt`), and moreover, if `a` is not `undef`, then we are adding an integer and a non-integer rational and the result is again `rationalNotInt` represented in the code by `b`.

- The case `a=rationalNotInt` and `b` a standard value different from `undef` is analogous to the previous one.

- Finally, if the type both parameters is not standard, this means that we are adding two non-integers rationals and the result is `undef` because the result can be either an integer (for instance $1/2 + 1/2$) or a non-integer rational ($1/2 + 1/3$).

The third redefinition of `+` corresponds to `intIrrat` values and is now easy to understand observing that:

- The code `(ca xor cb)` $\wedge$ `not(u)` $\wedge$ `r= irrational)` indicates that only one parameter is an `intRat` and, moreover, that this parameter is different from `undef`. In this case the other parameter is an irrational and the addition of a rational (either integer or not) and an irrational is always an irrational.

- The code `(not(ca)` $\wedge$ `not(cb))` $\wedge$ `r= undef)` indicates that if both parameters are irrational the result is undefined (we cannot be sure if the sum is rational or irrational).

*Appendix B.1.3. An Object-Oriented Perspective*

The idea of deriving new types by adding values to already existing types and redefining the behavior of certain operations, resembles loosely the principles of inheritance[19] applied in object-oriented programming. Therefore, it is worth exploring the relation between both approaches. Assume that class (respectively type) B is defined in terms of class (respectively type) A. Then:

- In object-oriented programming we say that *B is a subclass of A*, while in our approach we say that *B extends A*.
- In object-oriented programming B can *override* the methods of A under some circumstances. Analogously, in our approach B can *redefine* the operators of A.
- When overriding a method, B can use the definition of the same method in A. For instance, in Java this is done using the keyword `super`. In our approach this is done by using the **`prdf`** reserved word.

However, these similarities are misleading; extending is not a synonymous of subclassing. In fact, extending is closer to the concept of *superclassing* in object-oriented languages:

*In object-oriented programming, if B is derived from A, then B is a subclass of A, that is every B-object is considered an A-object, but not the other way round. On the contrary, in our approach if B extends A, then every A-value is a B-value, but not the other way round.*

For instance, if we define in an object-oriented language `intE` as a subclass of **`int`**, then it is assumed that every `intE` object can be considered an **`int`** object. However, in the example of Figure B.12 it is clear that every **`int`** value is in fact an `intE` value, but there is a `intE` value, namely `undef`, that is not in **`int`**.

This explains why before using **`prdf`** we ensure using the primitive **`sv`** that the operands correspond to standard values; we need to ensure that the operands belong to a 'subclass', which is not always true. Thus, **`prdf`** is not the analogous of `super` in Java, but the correspondence of a casting used to convert an object of class B to the superclass A. In Java such casts are usually preceded by some check like `if(x instanceof A) A y = (A)x;`. In our setting the implicit cast required by **`prdf`** is usually combined with a type check performed by the primitive **`sv`**.

### Appendix B.1.4. Implementation

Currently under development, the implementation of this more general setting is easy to describe as an iterative application of the transformation defined in Section 2. The idea is to apply the transformation starting from those extensions which corresponds to the leaves of the corresponding tree hierarchy. For instance, in the example of Figure B.12, first `intInf` e `intIrrat` are eliminated using the transformation of Section 2. Then, the transformation is again applied to `intRat`, and finally to `intE`, obtaining a standard and equivalent MiniZinc model.

A possible future line of research could be allowing to define new extensions using two (or more) base types. In this case instead of tree shaped hierarchies the structure would be a more general meet-semilattice[20], posing new problems similar to those of multiple inheritance in the case of object oriented programming. From the point of view of the program transformation a new variable for each base type will be required, thus modifying the transformation rules of Section 2.

### Appendix B.2. Extending Union Types

The extension of union types makes little sense in our approach. The main advantage of type extensions is to redefine the behaviour of operators defined on standard MiniZinc types, and there is no predefined operator on user types (except equality and disequality). Allowing the introduction of new operators over union types will introduce this possibility but defines a completely new framework.

Currently, if the modeller needs a new type including new constants, then they can define a new union type based on the previous one. For instance, consider this simple representation of natural numbers using its Peano representation:

```
enum nat = {zero ,s(nat)};
```

And suppose that the modeller later needs to introduce a new type that considers `null` as a possible value. Then a simple solution is:

```
enum nat = {zero ,s(nat)};
enum natnull = {notnull(nat), null};
```

### Appendix B.3. Union types based on Extended Types

This combination is possible, and indeed can be useful. Consider the following fragment of a model where a stack of extended integers is defined:

```
extended intInf = [negInf]++int++[posInf];
enum stack  = { empty, s(intInf,stack) };
var stack(3):a;
var stack(3):b;


% stack t = push element x on stack s
predicate push(var intInf:x, var stack:s, var stack: t) =
 t = s(x,s);

constraint push(posInf,empty,a);
constraint push(4,a,b);
...
```

It is interesting to observe the second constraint `push(4,a,b)`. In this predicate call `4` is an **`int`**, and `push` expects an `intInf` as first parameter. However, as explained above, **`int`** can be considered a subclass of `intInf` and thus `4` is also a `intInf`.

Regarding the implementation, it is obtained for free simply organizing the two phases of the transformation: first union types are eliminated following transformation of Section 3, keeping type extension declarations in the transformed model, and considering extension values and variables as standard types. The result is a MiniZinc$^\star$ model, which is then converted into standard MiniZinc using the transformation of Section 2.

*Appendix B.4. Mutually recursive Union Types*

Defining union types based on other union types, or even allowing recursive union types is sound, and in fact is included the default approach of Section 3. For instance the example of Figure 7 uses the recursive union type *tree*. Therefore the transformation of this mutually recursive, or self-recursive types has been already discussed during the presentation of these types.

## References

[1] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, Minizinc: Towards a standard CP modelling language, in: Proc. of 13th International Conference on Principles and Practice of Constraint Programming, Springer, 2007, pp. 529–543.

[2] IEEE Task P754, ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic, IEEE, 1985.

[3] F. Azevedo, Thesis: Constraint solving over multi-valued logics - application to digital circuits, AI Commun. 16 (2003) 125–127.

[4] E. F. Codd, Missing information (applicable and inapplicable) in relational databases, SIGMOD Record 15 (1986) 53–78.

[5] G. Malinowski, Many-Valued Logics, Oxford University Press, 1993.

[6] F. Barbanera, M. Dezani-Ciancaglini, U. de' Liguoro, Intersection and union types: syntax and semantics, Information and Computation 119 (1995) 202–230.

[7] P. J. Stuckey, G. Tack, Minizinc with functions, in: Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming, number 7874 in LNCS, Springer, 2013, pp. 268–283.

[8] A. Frisch, P. Stuckey, The proper treatment of undefinedness in constraint languages, in: I. Gent (Ed.), Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, volume 5732 of *LNCS*, Springer-Verlag, 2009, pp. 367–382.

[9] L. D. Koninck, S. Brand, P. J. Stuckey, Constraints in non-boolean contexts, in: Technical Communications of the 27th International Conference on Logic Programming, volume 11 of *LIPIcs*, 2011, pp. 117–127.

[10] R. Caballero, P. J. Stuckey, A. Tenorio-Fornés, Finite type extensions in Constraint Programming (extended version), Technical Report SIC-05/13, Facultad de Informática, Universidad Complutense de Madrid, 2013. `http://gpd.sip.ucm.es/rafa/minizinc/cptr.pdf`.

[11] R. Caballero, J. Luzón-Martín, A. Tenorio-Fornés, Test-Case generation for SQL nested queries with existential conditions, Electronic Communications of the European Association for the Study of Science and Technology 55 (2012).

[12] E. F. Codd, Extending the database relational model to capture more meaning, ACM Transactions on Database Systems 4 (1979) 397–434.

[13] Gecode Team, Gecode: Generic Constraint Development Environment, 2006. Available from `http://www.gecode.org`.

[14] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, M. Wallace, The design of the Zinc modelling language, Constraints 13 (2008) 229–267.

[15] L. De Koninck, S. Brand, P. Stuckey, Data independent type reduction for Zinc, in: T. Mancini, J. Pearson (Eds.), Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010), 2010, p. 15.

[16] C. Mears, A. Schutt, P. J. Stuckey, G. Tack, K. Marriott, M. Wallace, 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, number 8451 in LNCS, Springer, 2014, pp. 88–103. doi:`http://dx.doi.org/10.1007/978-3-319-07046-9_7`.

[17] P. Laborie, J. Rogerie, Reasoning with conditional time-intervals, in: D. C. Wilson, H. C. Lane (Eds.), Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, AAAI Press, 2008, pp. 555–560.

[18] M. Gavanelli, E. Lamma, P. Mello, M. Milano, Dealing with incomplete knowledge on clp(fd) variable domains, ACM Trans. Program. Lang. Syst. 27 (2005) 236–263.

[19] B. Meyer, Object-Oriented software construction, second Edition ed., Prentice Hall, 1997.

[20] B. A. Davey, H. A. Priestley, Introduction to lattices and order / B.A. Davey, H.A. Priestley, Cambridge University Press Cambridge [England] ; New York, 1990.