Finite Type Extensions in Constraint Programming

Rafael Caballero
University Complutense of Madrid

Peter J. Stuckey
NICTA and the University of Melbourne

Antonio Tenorio-Fornés
University Complutense of Madrid

ABSTRACT

Many problems are naturally modelled by extending an existing type with additional values. For example for modelling database problems with nulls natural models use booleans and integers with an additional null value. Similarly models involving integers may naturally be extended to handle $-\infty$ and $+\infty$. We extend the constraint modelling language MiniZinc to MiniZinc$^+$ to allow modelling with extended types. 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. The resulting MiniZinc$^+$ model is transformed to a MiniZinc model which is equivalent to the original model. We illustrate the usage of MiniZinc$^+$ to model SQL like problems with integer variables extended with NULL values.

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 a technique for expressing constraints over extended types in the constraint modelling language MiniZinc [11].

For example, within our 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 result of the addition of two `intE` variables `x`, `y` as `x+y` if both `x` and `y` are in the subtype **int**, or `posInf` if at least one of the two values is `posInf` (as in IEEE standard 754 [7]).

Apart from extended arithmetic, the extension of standard domains is an approach used in a multitude of disciplines, such as the design and test of digital circuits [1], the representation of `null` values to represent the unknown data in database query languages such as SQL [5], or the many-valued logics [10]. All these problems can be successfully modeled in the language proposed in this paper, which we call MiniZinc$^+$.

In order to solve the constraints over the extended types we present a transformation from MiniZinc$^+$ into MiniZinc. The transformation represents each extended decision variable as a pair of variables in MiniZinc. The first variable contains a possible value in the source, 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 for constraint satisfaction problems, but also for optimization problems.

The next section introduces both the syntax of MiniZinc with functions [12] and the syntax of MiniZinc$^+$. Section 3 explains that the transformation is the composition of two phases. The first phase, the elimination of local declarations and functions is described in other papers and is not discussed here. The second part is itself split into two sections: first, Section 4 introduces the transformation over expressions, and then Section 5 generalizes the transformation to top-level constructions such as constraints and declarations. A working prototype following these ideas is presented in Section 6. The soundness of the approach is discussed in Section 7. Finally, Section 8 presents the conclusions and discusses possible future work.

2. EXTENDING MINIZINC

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 [12]. The grammar of MiniZinc$^+$, the extension proposed in this paper, is basically the grammar of MiniZinc adding only the possibility of declaring new, extended types:

$$
\begin{array}{rcl}
\texttt{typeE} & \longrightarrow & \textbf{extended}\ \texttt{tId} = \\
& & [\texttt{c}_{-n}, \ldots, \texttt{c}_{-1}]\ \texttt{++type++}\ [\texttt{c}_1, \ldots, \texttt{c}_m] \\
\texttt{exp} & \longrightarrow & \texttt{vId} \mid \texttt{constant} \mid \texttt{vId[exp]} \\
& & \mid \texttt{arrexp[exp]} \mid \texttt{setexp} \mid \texttt{arrexp} \\
& & \mid \textbf{if}\ \texttt{exp}\ \textbf{then}\ \texttt{exp}\ \textbf{else}\ \texttt{exp}\ \textbf{endif} \\
& & \mid \texttt{pId(exp}^{*[,]}\texttt{)} \mid \texttt{fId(exp}^{*[,]}\texttt{)} \\
& & \mid \textbf{let}\ \{\texttt{decl}^{*[;]}\ \texttt{const}^{*[;]}\}\ \textbf{in}\ \texttt{exp} \\
& & \mid \textbf{forall}\ \texttt{(arrexp)} \\
& & \mid \textbf{exists}\ \texttt{(arrexp)}
\end{array}
$$

```
arrexp   ⟶  [exp*[,]]
             | [exp | genvar+[,] where exp]
setexp   ⟶  { exp*[,] } | range
             | {exp | genvar+[,] where exp}

genvar   ⟶  vId+[,] in setexp | vId+[,] in arrexp
range    ⟶  exp .. exp
decl     ⟶  vtype : vId
             | array[range] of vtype : vId
             | set of type: vId
             | var set of setexp: vId
assig    ⟶  vId = exp

const    ⟶  constraint exp
funct    ⟶  function decl (decl*[,]) = exp
pred     ⟶  predicate pId(decl*[,]) = exp

solv     ⟶  solve satisfy | solve minimize vId
             | solve maximize vId
out      ⟶  output ([ sh*[,] ])
sh       ⟶  show(exp) | "string"
type     ⟶  int | bool | float | tId | range
vtype    ⟶  type | var type
model    ⟶  typeE*[;];decl,*[;]; assig*[;]; pred*[;]
             ; funct*[;];const*[;]; solv; out;
```

where `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. **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 [12] is the new nonterminal `typeE` and the inclusion of type identifiers (`tId`) as possible types.

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 [1]. Then, in our setting we can extend the standard MiniZinc Boolean type a new constant `undef`. The definition in MiniZinc$^+$ of the new type can be found in the first line of the model in Figure 1. Note that replacing `bEx` with **bool** in lines (3-6) and omitting lines (8-28) would give 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 2 (where 0 stands for **false**, 1 for **true** and $\perp$ stands for `undef`). For instance, the standard MiniZinc operator **xor** is redefined in MiniZinc$^+$ as shown in lines (8-12) of Figure 1. 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, then the function returns the result of using the standard MiniZinc operator **xor**. Otherwise, if either `a` or `b` is `undef`, then the result is `undef` according to

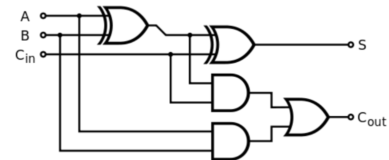
```
1  extended bEx = bool ++ [undef];
2  int n;
3  array[1..n]   of var bEx: x;
4  array[1..n]   of var bEx: y;
5  array[1..n+1] of var bEx: s;
6  array[1..n+1] of var bEx: c;
7
8  function var bEx:xor(var bEx:a, var bEx:b) =
9    let{var bEx:r, var bool:c1=sv([a,b]),
10       constraint (c1 /\ (r= a xor b)) \/
11                  (not c1 /\ r=undef)
12   } in r;
13
14 function var bEx:/\(var bEx:a, var bEx:b) =
15  let{var bEx:r, var bool:c1=sv([a,b]),
16      var bool:c2= (a=false \/ b=false),
17      constraint (c1 /\ r=a /\ b) \/
18                 (not c1 /\ c2 /\ r=false) \/
19                 (not c1 /\ not c2 /\ r= undef)}
20      in r;
21
22 function var bEx:\/(var bEx:a, var bEx:b) =
23  let{var bEx:r, var bool:c1=sv([a,b]),
24      var bool:c2= (a=true \/ b=true),
25      constraint (c1 /\ r= a \/ b) \/
26                 (not c1 /\ c2 /\ r=true) \/
27                 (not c1 /\ not c2 /\ r=undef)}
28      in r;
29
30 constraint c[1]=false /\ s[n+1]=c[n+1]
31 constraint forall([s[i]=x[i] xor y[i] xor
32                c[i]|i in 1..n])
33 constraint forall([c[i+1]=(x[i] /\ y[i]) \/
34               ((x[i] xor y[i]) /\
35                  c[i])|i in 1..n]);
36 solve satisfy;
```

Figure 1: An n bit full adder in MiniZinc$^+$: $x + y = s$

the table for extended **xor** of Figure 2. The schema of this function will be usual in all the `conservative` redefinition of standard operators. The code for functions redefining $\wedge$ and $\vee$ is analogous.

Note that although the functions **xor**, $\wedge$ and $\vee$ have been redefined, they are used as the original functions inside function declarations (since they apply to the original type **bool**).

Using these definitions we model the behavior of an n-bit adder digital circuit in lines (30-35). The basic piece of the circuit is the *full adder*:



which adds binary numbers and accounts for values carried

	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 2: Truth tables including the undefined value

in as well as out. The code of lines (30-35) employs n full adders to obtain an n-bit adder. In particular, line (32) defines the output s using two *xor* gates, while lines (33-35) 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:[1]

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

The least significant digit (and thus the first position of each array) is displayed on the left. Observe that in the second position from the left the addition $\perp + \perp +1$ (1 is the carry from the previous 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 ensure that the carry is 0, and thus in the fourth position we have $1 + 0 + 0 = 0$ as output with 0 carry and as last bit.

3. FROM MINIZINC+ TO MINIZINC

The main goal of this paper is to present an automatic translation from MiniZinc$^+$ 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. Finally, the resulting MiniZinc$^+$ 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$^+$ indistinctly. In particular, the function elimination is done unrolling the function calls following ideas similar to those described in [12] (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 [6] of MiniZinc where these constraints "float" to the nearest enclosing Boolean context where they are added as a conjunct. Analogously, the local variable declarations are converted to global variable declarations, see [8] for a more detailed discussion.

[1]The **`output`** sentence is omitted in Figure 1 for simplicity.

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

4. TRANSFORMING MINIZINC+ EXPRESSIONS

In the case of MiniZinc$^+$ 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 $\tau_e(c)$).

4.1 Notation

First we introduce some auxiliary notation:

We use t for type identifiers (either standard as **`bool`**, **`int`** and **`float`** or extended such as `boolEx`). The functions $st(t)$ and $et(t)$ return whether t is either a standard (st) or an extended (et) type.

The notation $\texttt{ord}_t\texttt{(k)}$ maps constants `k` of type t to 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:

- `ord(negInf) = -1`
- `ord(undef)= 1`
- `ord(posInf) = 2`

For every constant k, $\texttt{ord}_t\texttt{(k)} \neq 0$ iff `k` is extended. We define $\texttt{ord}_t(k) = 0$ if k is a standard constant. The function $\texttt{eRan}(t)$ (extended Range) is defined for an extended type t as follows: define a set S as $S = \{ord_t(k)|k \in t\} \cup \{0\}$, then $eRan(t) = $ **`min`**`(S) ..` **`max`**`(S)`. In the example of `int3` above: -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$^+$ models are well-typed following the type inference rules for MiniZinc which can

be found in [3], and use the notation *type(e)* to refer to the type of e.

Next we explain the transformation of MiniZinc^{+} expressions, distinguishing between the different possibilities enunciated in the grammar (Section 2.1).

4.2 Identifiers, constants, array and set expressions

Identifiers and constants. The transformations τ_s and τ_e for identifiers and constants are defined as follows:

	τ_s	τ_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)	$k_{o(t)}$	$ord_t(k)$

Observe that here identifiers represent both decision variables and parameters. Identifiers of standard type are mapped to the original form, with the second component fixed to zero, representing a standard value. Extended type identifiers are mapped to the associated new identifiers. 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.

Array expressions. Array expressions of the form: $e = [e_1, \ldots, e_n]$ are transformed simply mapping the transformations τ_s, τ_e:

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

For instance, if we consider the array expression e defined as [*true,false,undef*], then $\tau_s(e)$ = [*true,false,false*], and $\tau_e(e)$ = [*0,0,1*]. Observe that the underlined *false* corresponds to the arbitrary constant $k_{Boolean}$ 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 access of the form $a[exp]$ with $type(a)$ = *<array of t>* is transformed as:

τ_s	τ_e
τ_s(a)[τ_s(exp)]	τ_e(a)[τ_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 1. We have c = *<array of boolEx>*, and thus *st(boolEx)* is false and *et(boolEx)* 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`.[2]

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

Set expressions. Set expressions of the form $e = \{ e_1, \ldots, e_n \}$ with

$type(e_1) = \cdots = type(e_n) = t$ are transformed depending on the type t:

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

  $\tau_s(e)$ = {[$\tau_s(e_1), \ldots, \tau_s(e_n)$][i] | i in 1..n **where** [$\tau_e(e_1), \ldots, \tau_e(e_n)$][i] = 0 }

  $\tau_e(e)$ = {[$\tau_e(e_1), \ldots, \tau_e(e_n)$][i] | i in 1..n **where** [$\tau_e(e_1), \ldots, \tau_e(e_n)$][i] != 0 }

The overall idea is that the elements in the set are split into standard and extended parts.

4.3 Array and set comprehensions

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

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

  [ord_t^{-1}(x) | x **in** τ_e(setexp) **where** x<0]++

  [x | x **in**τ_s(setexp)]++

  [ord_t^{-1}(x) | x **in** τ_e(setexp) **where** x>0].

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

Let ⟨`(exp')` |`genvars'` **where** `cond'` ⟩ 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:

τ_s = [τ_s(exp') | genvars' **where** τ_s(cond')]

τ_e = [τ_e(exp') | genvars' **where** τ_s(cond')]

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

τ_e = { τ_e(exp') | genvars' **where** τ_s(cond') $\wedge$ τ_e(exp)!=0 }

For example, let *intE* be the integer type extended with 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 `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 τ_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]!=0]

Then the array a is defined as:

a = [ord$_t^{-1}$(x) | x **in** τ_e(S) **where** x $<$ 0] ++
[x | x **in** τ_s(S)] ++
[ord$_t^{-1}$(x) | x **in** τ_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:

$\tau_s(e)$ = [τ_s(a[j])| i **in** **index-set**([0,4,9,-1]),
j **in** **index-set**(τ_s(a))
where τ_s(L[i]=a[j])]

and

$\tau_e(e)$ = [τ_e(a[j])| i **in** **index-set**([0,4,9,-1]),
j **in** **index-set**(τ_s(a))
where τ_s(L[i]=a[j])]

During the evaluation the system will obtain:

τ_s(e) = [0,-1], and τ_e(e) = [1,0],

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

4.4 Conditional and logical expressions

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

τ_s(e) = **if** τ_s(c) **then** τ_s(e1) **else** τ_s(e2) **endif**

τ_e(e) = **if** τ_s(c) **then** τ_e(e1) **else** τ_e(e2) **endif**

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

4.5 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: τ_s(c) = (τ_e(exp_1)=z) $\wedge \cdots \wedge$ (τ_e(exp_n)=z), with z the value zero value associated to the type of the expressions.

- c $\equiv$ predef(f)(exp_1, ..., exp_n), or alternatively c $\equiv$ exp_1 predef(f) exp_2, with f a predefined function or an infix operator. `predef` indicates that this call corresponds to the predefined MiniZinc function/operator f even if it has been redefined by the user. Thus, τ_s(c)= f(τ_s(exp_1), ..., τ_s(exp_n)), or τ_s(c)=τ_s(exp_1) f τ_s(exp_2) if f is an infix operator, and τ_e(c) = z. 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: τ_s(c) is defined as

(τ_s(exp_1) = τ_s(exp_2) $\wedge$$\tau_e$($\text{exp}_1$) =$\tau_e$($\text{exp}_2$))

and τ_e(c) is defined as z. 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). The translation of exp_1 != exp_2 is simply not(exp_1= exp_2), applying then the translation of =.

- $c \equiv (e\ in\ S)$, assuming that *in* has not been redefined. Then: $\tau_s(c) = (\tau_e(e) = 0 \wedge \tau_s(e)\ in\ \tau_s(S)) \vee (\tau_e(e) \neq 0 \wedge \tau_e(e)\ in\ \tau_e(S))$ 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.

5. TRANSFORMING MINIZINC+ MODELS

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

5.1 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.

5.2 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 $c^{\mathcal{T}} \equiv \tau(decl)$ as defined by the following table:

	τ
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** τ_s(setexp): s(x); **var set of** τ_e(setexp) : e(x)

with the constraints C_1 and C_2 defined as

$C_1 \equiv$ **constraint** `xe!=`$z_{o(t)}$ `->` `xs=`$k_{o(t)}$`;`

and

$C_2 \equiv$ **constraint** **forall**`([ae[i]!=`$z_{o(t)}$`->` `as[i]=`$k_{o(t)}$ `| i` **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 $\neq z$), then the standard part of the variable takes some arbitrary value k_t.

In our running example, the array `ia` is transformed into:

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

assuming that **false** is the arbitrary constant k_{bool}.

5.3 Assignments and Constraints

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

	τ
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: $c^{\mathcal{T}} =$ **constraint** τ_s(exp).

5.4 Output Item

The translation of an output item adds a new requirement, being able to print extended types. An expression **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[`τ_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 ];
```

```
1 extended time = (0..23) ++ [oneDayOrMore];
2
3 function var time:+(var time:x, var time:y) =
4  let {var time:r, var bool:c=sv([x,y]),
5      constraint
6       (c /\ x + y>23 /\ r=oneDayOrMore) \/
7       (c /\ x + y<=23 /\ r=x+y ) \/
8       (not c /\ r=oneDayOrMore) }
9       in r;
10
11 time: t1 = 5;
12 var time:t2;
13 var time:total = t1 + t2 + 21;
14 solve minimize total;
15 output(["Total=",show(total),
16         " t2=",show(t2),"\n"]);
```

Figure 3: Modelling time with an extended value..

5.5 Satisfaction and Optimization

A *satisfaction problem* is encoded in MiniZinc+ using the solve item **solve satisfy**. In the translation to MiniZinc this is unchanged.

MiniZinc also allows defining *optimization problems*, using **solve minimize** e or **solve maximize** e. In MiniZinc+ 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 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 dont exist then we may be able to determine $a = \min(\tau_s(y))$ 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.[3] Given a and b can be determined we transform minimize/maximize y to minimize/maximize $\tau_e(y) * (b - a + 1) + \tau_s(y)$.

For instance, the example in Figure 3 models the time required to perform some task. The time is measured in hours, from 0 to 23, plus an especial 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 $b = 23$.

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

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 c:

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

6. EXPERIMENTAL RESULTS

This section presents a practical example of usage MiniZinc+ that has been used to check the feasibility of the proposal from the point of view of the implementation.

The tool *STCG* [2] generates MiniZinc models whose solutions constitute test cases for testing SQL views. However, the constraints generated do not consider *NULL* values, an important feature in the relational database model[4].

In order to allow NULL values a part of the possible test cases we extend the models including two new types: [4]

```
extended intE = [] ++ int ++ [NULL];
extended boolEx = [] ++ bool ++ [NULLb];
```

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

```
function var boolEx: '='(var intE:x, var intE:
    y) =
  let {
    var boolEx: r,
    var bool: c1,
    constraint c1 = (sv(x) predef(/\) sv(y)),
    constraint
      (not(c1) predef( /\ ) eq(r,NULLb))
     predef( \/ )
      ( c1  predef( /\ ) eq(r, (x = y)))
  } in r;

function var boolEx: '!='(var intE:x, var intE
    :y) =
  let {
  var boolEx: r,
  var bool: c1,
  constraint eq(c1,(sv(x) predef(/\) sv(y))),
  constraint
      (not(c1) predef( /\ ) eq(r, NULLb))
     predef( \/ )
      ( c1  predef( /\ ) (r predef(=) not (x
          predef(=) y)))
    } in r;

function var boolEx:'/\'(var boolEx:a1, var
    boolEx:b1) =
  let{var boolEx:r1,
    var bool:c11,
    var bool:c21,
    constraint (c11  predef(=) (sv(a1) predef(
         /\ ) sv(b1))),
    constraint (c21  predef(=) (eq(a1,false)
        predef( \/ ) eq(b1,false))),
```

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

Model	*Sql Or*	*Sql Or*$^+$	*Board*	*Board*$^+$
var. decl.	5	99	54	2032
funct. calls	4	254	419	374678
size (KB)	0.5	5.0	13.2	15946.2
transf. time		17		2923
solve time	0.50	0.29	0.32	2.21

Table 1: Experimental data for two models generated by STCG

```
    constraint (c11 predef( /\ ) eq(r1,(a1
        predef( /\ ) b1)))
      predef( \/ )
      (not (c11) predef( /\ )  c21 predef( /\
         )  eq(r1,false))
      predef( \/ )
      (not (c11) predef( /\ ) not (c21) predef
         ( /\ )  eq(r1, NULLb))
    } in r1;

function var boolEx:'\/'(var boolEx:aa, var
    boolEx:bb) =
  let{var boolEx:rr,
    var bool:cc1,
    var bool:cc2,
    constraint (cc1 predef(=) (sv(aa) predef(
        /\ ) sv(bb))),
    constraint (cc2  predef(=) ((eq(aa,true)
        predef( \/ ) eq(bb , true)))),
    constraint
      (cc1  predef( /\ ) eq(rr, (eq(aa,true)
          predef( \/ ) eq(bb, true))))
      predef( \/ )
      (not (cc1) predef( /\ ) cc2 predef( /\ )
          eq(rr, true)) predef( \/ )
      (not (cc1) predef( /\ ) not (cc2) predef
         ( /\ ) eq(rr,NULLb))
    } in rr;
```

Adding this code to the models produced by *STCG* and changing the type of its variables to the new extended types, we have automatically test-cases including NULL values. In order to check the efficiency of the proposal we have tried two different examples of models produced by *STCG* (see [3] for a detailed description of the two examples).

Table 1 shows the data obtained with our current implementation. The two models in MiniZinc produced by *STCG* are called *Sql Or* and *Board*. The MiniZinc models obtained after introducing the new type, redefining the operators and applying the transformation described in this paper are called in the table *Sql Or*$^+$ and *Board*$^+$.

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 2032 variables in the model when considering NULL values.
- *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$^+$ to support NULLs and applying the transformation to obtain the equivalent MiniZinc model we obtain a model with 254 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$^+$ model into a MiniZinc model.
- *solve time*: The time required by the MiniZinc solver to obtain the first answer.

Observe that although the transformation increases the size, number of variables or function calls of the transformed model, this overload in size does no imply a very significative increment in time. This indicates that the theoretical proposal can be used in practice.

The prototype can be downloaded from `http://gpd.sip.ucm.es/rafa/minizinc/extendedMiniZinc.tar.gz`

7. THEORETICAL RESULTS

In this section we present the theoretical result that supports our proposal. The idea is to prove that both the MiniZinc$^+$ 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$^+$ model, Γ its associated type context, and θ a substitution. We say that θ is a* well-typed *substitution for $\mathcal{M}$ iff*

- *The domain of θ is the set containing the decision variables declared in $\mathcal{M}$.*[5]
- *For all $x \in dom(\theta)$, $type(x) =< t >$ iff $type(x\theta) =< t >$*

The key idea for defining the concept of solution is the evaluation of an expression in a model with respect to a given well-typed substitution.

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

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

1. $\| id \|_\theta = id\theta$, id *any identifier.*

2. $\| k \|_\theta = k$, k *any constant.*

3. *Set Expressions:*

   (a) $\| \{e_1, \ldots, e_n\} \|_\theta = ord(\{\| e_1 \|_\theta, \ldots, \| e_n \|_\theta\})$. *ord is defined as the function that given a set of values, eliminate the repetitions and sort the values according to order* $\preceq$ *that extends* ord_t *defined in Section 4.1 where:*

$$a \preceq b = \begin{cases} a \leq b & a, b \text{ standard} \\ ord_t(a) < 0 & a \text{ ext.}, b \text{ std.} \\ ord_t(b) > 0 & a \text{ std.}, b \text{ ext.} \\ ord_t(a) \leq ord_t(b) & \text{otherwise} \end{cases}$$

   (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 a an array identifier with index range* $m \ldots n$, $i = \| e \|_\theta - m + 1$, $1 \leq i \leq n - m + 1$, *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* $i = \| e_2 \|_\theta$, $1 \leq i \leq n$.

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:*

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

   (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:*
   $\| \langle e \mid g_1, \ldots, g_m \text{ where } c \rangle \|_\sigma =$
   $\| \langle e \mid g_2 \ldots, g_m \text{ where } c \rangle \|_{\sigma_1} \ \diamond \ldots \diamond$
   $\| \langle e \mid g_2 \ldots, g_m \text{ where } c \rangle \|_{\sigma_n}$

7. $\| sv([e_1, \ldots, e_n]) \|_\theta = st(t_1) \wedge \cdots \wedge st(t_n)$ *with* $\Gamma \vdash \| e_1 \|_\theta :: t_1, \Gamma \vdash \| e_n \|_\theta :: t_n$

8. $\| e_1 = e_2 \|_\theta = true$ *if* $\| e_1 \|_\theta$ *and* $\| e_2 \|_\theta$ *are the same constant, false otherwise.*

9. $\| p(e_1, \ldots, e_n) \|_\theta = p(\| e_1 \|_\theta, \ldots, \| e_n \|_\theta)$ *, with p* MiniZinc *predefined (that p is a relational operator or predefined arithmetic function such as* >,<,+ ...*) .*

10. Forall, exists *constructions:*
    Let $\| a \|_\theta$ *be* $[v_1, \ldots, v_n]$, *then:*

    - $\| forall(a) \|_\theta = v_1 \wedge \cdots \wedge v_n$
    - $\| exists(a) \|_\theta = v_1 \vee \cdots \vee v_n$

Thus, the overall idea is simply to evaluate the expressions after replacing the variables by their values. Now we can define the concept of solution.

Definition 3. *Let* $\mathcal{M}$ *be a MiniZinc*$^+$*model defined by* $\mathcal{M} = T; D; A; P; F; C; S$, *with T the sequence of type extensions declarations, D the sequence of declarations, A the sequence of assignments, C the sequence of constraints, and S the* solve *statement. Let* θ *be a well-typed substitution for* $\mathcal{M}$. *Then, we say that* θ *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* σ *for* $\mathcal{M}$ *verifying 1) and 2) and such that* $f\sigma > f\theta$ *(respectively* $f\sigma < f\theta$*)*

Definition 4. *Let* $\mathcal{M}$ *be a MiniZinc*$^+$*model and* σ *be a well-typed substitution of* $\mathcal{M}$, *then,*

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

Finally, we can establish the theoretical result.

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

Proof Idea

We must check that both θ verifies the three items of Definition 4 with respect to $\mathcal{M}$ iff $\theta^{\mathcal{T}}$ verifies the same Definition with respect to $\mathcal{M}^{\mathcal{T}}$.

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 θ*:*

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

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. A detailed proof can be found in [3].

8. CONCLUSIONS AND FUTURE WORK

The possibility of extending predefined types with new constants allows the representation of many constraint satisfaction problems in a more natural way. Some examples are models representing circuits including undefined entries (representing for instance failing connections), database problems including *null* values, problems that can be modelled using many-valued logics, or scheduling problems with optional tasks.[6] Clearly the modeller could directly use MiniZinc rather than MiniZinc$^+$ to model their problem (since MiniZinc$^+$ is 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.

The system MiniZinc$^+$ presented in this paper extends the constraint system MiniZinc to include this feature. The modeller can define new types by adding new constants to already existing types, and redefine accordingly the behaviour of the predefined operations. We present a model transformation that converts the models in the new system into a standard MiniZinc model. Thus, all the facilities included in MiniZinc such as intensional lists, local definitions, sets, or predicates are available in the new setting. The proposal has been implemented in a working prototype.

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

As future work we plan to allow the possibility of extending already extended types. The framework will give rise to lattices of extensions and will allow modelling more complex problems.

[6] Although for these scheduling problems there are approaches [9] which support stronger propagation.

9. REFERENCES

[1] F. Azevedo. Thesis: Constraint solving over multi-valued logics - application to digital circuits. *AI Commun.*, 16(2):125–127, 2003.

[2] R. Caballero, J. Luzón-Martín, and A. Tenorio-Fornés. Test-Case Generation for SQL Nested Queries with Existential Conditions. *Electronic Communications of the EASST*, 55, 2012.

[3] R. Caballero, P. J. Stuckey, and 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`.

[4] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, Dec. 1979.

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

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

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

[8] L. D. Koninck, S. Brand, and P. J. Stuckey. Constraints in non-boolean contexts. In *ICLP (Technical Communications)*, pages 117–127, 2011.

[9] P. Laborie and J. Rogerie. Reasoning with conditional time-intervals. In D. C. Wilson and H. C. Lane, editors, *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, pages 555–560. AAAI Press, 2008.

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

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

[12] P. J. Stuckey and 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*, LNCS, page to appear. Springer, 2013.