Finite Type Extensions in Constraint Programming (extended version)

Rafael Caballero[1], Peter J. Stuckey[2] and Antonio Tenorio-Fornés[1]

Technical Report SIC-05-13

[1] University Complutense of Madrid
[2] NICTA and the University of Melbourne

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 integers with an additional null value. Similarly models involving integers may naturally be extended to handle $-\infty$ and $+\infty$. We extend 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 model MiniZinc$^+$ model is transformed to a MiniZinc model which is equivalent to the original model. We illustrate the usage of MiniZinc$^+$ to model Boolean circuits allowing undefined inputs and scheduling problems considering special time 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 [9].

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 [5]).

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 [3], or the many-valued logics [8]. 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 the possible value in the source, standard type. The second variable contains the 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 [10] 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 decribed in other papers and is not discussed here. The second part is itself split into two section: first, Section 4 introduces the transformation over expressions, and then Section 5 generalizes the transformation to top-level constructions such as constraints and declarations. The soundness of the approach is discussed in Section 7. A working prototye following these ideas is presented in Section ?? Finally, Section 9 presents the conclusions and discusses possible future work.

2 Extending MiniZinc

2.1 Syntax

In this section we introduce the syntax of MiniZinc$^+$, our proposed extension of MiniZinc (with functions) [10]:

$$
\begin{array}{rcl}
\text{typeE} & \longrightarrow & \textit{extended}\ \mathbf{tId} = [c_{-n}, \ldots, c_{-1}] \mathbin{++} \text{type} \mathbin{++} [c_1, \ldots, c_m] \\
\text{exp} & \longrightarrow & \mathbf{vId} \mid \mathbf{constant} \mid \mathbf{vId}[\text{exp}] \mid \text{arrayexp}[\text{exp}] \mid \text{setexp} \mid \text{arrayexp} \\
 & \longrightarrow & \mid \textit{if}\ \text{exp}\ \textit{then}\ \text{exp}\ \textit{else}\ \text{exp}\ \textit{endif} \\
 & \longrightarrow & \mid \mathbf{pId}(\text{exp}^{*[,]}) \mid \mathbf{fId}(\text{exp}^{*[,]}) \mid \textit{let}\ \{\text{decl}^{*[,]}\ \text{const}^{*[,]}\}\ \textit{in}\ \text{exp} \\
 & \longrightarrow & \mid \textit{forall}\ (\text{arrayexp}) \mid \textit{exists}\ (\text{arrayexp}) \\
\text{arrayexp} & \longrightarrow & [\text{exp}^{*[,]}] \mid [\text{exp} \mid \text{genvar}^{+[,]}\ \textit{where}\ \text{exp}] \\
\text{setexp} & \longrightarrow & \{\ \text{exp}^{*[,]}\ \} \mid \text{range} \mid \{\text{exp} \mid \text{genvar}^{+}\ \textit{where}\ \text{exp}\} \\
\text{genvar} & \longrightarrow & \mathbf{vId}^{+[,]}\ \textit{in}\ \text{setexp} \mid \mathbf{vId}^{+[,]}\ \textit{in}\ \text{arrayexp} \\
\text{range} & \longrightarrow & \text{exp} \mathrel{..} \text{exp} \\
\text{decl} & \longrightarrow & \text{vtype} : \mathbf{vId} \mid \textit{array}[\text{range}]\ \textit{of}\ \text{vtype} : \mathbf{vId} \\
 & \longrightarrow & \mid \textit{set of}\ \text{type}\text{:}\ \mathbf{vId} \mid \textit{var set of}\ \text{setexp}\text{:}\ \mathbf{vId} \\
\text{assig} & \longrightarrow & \mathbf{vId} = \text{exp} \\
\text{const} & \longrightarrow & \textit{constraint}\ \text{exp} \\
\text{funct} & \longrightarrow & \textit{function}\ \text{decl}\ (\text{decl}^{*[,]}) = \text{exp} \\
\text{pred} & \longrightarrow & \textit{predicate}\ \mathbf{pId}(\text{decl}^{*[,]}) = \text{exp} \\
\text{solv} & \longrightarrow & \textit{solve satisfy} \mid \textit{solve minimize}\ \text{vId} \mid \textit{solve maximize}\ \text{vId} \\
\text{out} & \longrightarrow & \textit{output ([}\ \text{show}^{*[,]}\ \textit{])} \\
\text{show} & \longrightarrow & \textit{show(}\mathbf{vId}\textit{)} \mid \textit{show(}\text{string}\textit{)} \\
\text{type} & \longrightarrow & \textit{int} \mid \textit{bool} \mid \textit{float} \mid \mathbf{tId} \mid \text{range} \\
\text{vtype} & \longrightarrow & \text{type} \mid \textit{var}\ \text{type} \\
\text{model} & \longrightarrow & \text{typeE}^{*[;]}\text{;decl}^{*[;]}\text{; assig}^{*[;]}\text{; pred}^{*[;]}\text{; funct}^{*[;]}\text{; const}^{*[;]}\text{; solv; out}
\end{array}
$$

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. The values c_i represent new constant identifiers. The notation $\text{n}^{*[s]}$ / $\text{n}^{+[s]}$ indicates zero or more / one or more repetitions of the nonterminal n such that these repetitions are separated by string s. *Italic* words are reserved words of the language. The only difference of this grammar with respect to the standard MiniZinc with functions presented in [10] 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 extend the Boolean type with a new constant *undef* in order to model combinational circuits with undefined (i.e. neither true or false) signals [1]. The definition in MiniZinc$^+$ of the new type can be found in the first line of the model in Figure 1. Note that replacing `boolEx` with `bool` in lines (3-5) and omitting lines (8-26) would give a standard MiniZinc model for this problem.

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

Fig. 1: A n bit full adder in MiniZinc$^+$: $x + y = s$

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 (8-11) 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 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.

	1	**0**	**⊥**
1	1	1	1
0	1	0	⊥
⊥	1	⊥	⊥

(a) ∨

	1	**0**	**⊥**
1	1	0	⊥
0	0	0	0
⊥	⊥	0	⊥

(b) ∧

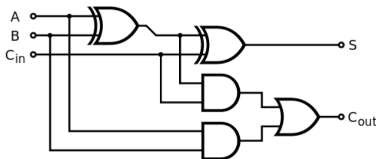
	1	**0**	**⊥**
1	0	1	⊥
0	1	0	⊥
⊥	⊥	⊥	⊥

(c) xor

Fig. 2: Truth tables including the undefined value

Note that although the functions xor, ∧ and ∨ 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 a n-bit adder digital circuit in lines (26-30). 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 (25-28) employs n full adders to obtain a n-bit adder. In particular, line (26) defines the output s using two *xor* gates, while lines (27-28) model the carries employing two *and* and one *or* gates.

After transforming this model into an standard MiniZinc model, we can use MiniZinc to obtain solutions such as the following:[3]

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

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 ⊥ + ⊥ +1 (1 is the carry from the previous position) yields ⊥ 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 ⊥. 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 ex-

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

tended 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 [10] (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 [4] 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 [6] for a more detailed discussion.

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, we define two different 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 $\mathsf{ord}_t(k)$ 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 subindex t in ord is omitted when it is clear from the context). For instance, given the definition

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

we have ord(negInf) $= -1$, ord(undef)$=1$ and ord(posInf)$=2$. For every constant k, ord_$t(k) \neq 0$ iff k is extended. We define $\mathsf{ord}_t(k) = 0$ if k is a standard constant. The function 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 [2], and use the notation *type(e)* to refer to the type of e.

Next we explain the transformation of (extended) MiniZinc$^+$ expressions, distinguishing between the different possibilities enunciated in grammar of 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:

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

For instance, if *e=[true,false,undef]*, then $\tau_s(e) =$ [true,false,<u>false</u>], 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 25 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.[4]

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 $\tau_s(genExp)$].
– 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:

[4] 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.

– 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 let L be [posInf, 4, 9, -1], and let S be 8, -1, 8, posInf. Therefore, the array comprehension is represented as [y | x in L, y in L 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 [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] and $\tau_e(S)$ = [[0,0,0,1][i] | i in 1..4 where [0,0,0,1]!=0]. Then the array a is defined as:

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

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 mimicks 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(τ_s(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])]
$\tau_e(e)$ = [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

Conditional expressions of the form $e \equiv$ if c then e_1 else e_2 *endif* are transformed as:

- $\tau_s(e)$ = if $\tau_s(c)$ then $\tau_s(e_1)$ else $\tau_s(e_2)$ endif
- $\tau_e(e)$ = if $\tau_s(c)$ then $\tau_e(e_1)$ else $\tau_e(e_2)$ endif

The exists and forall constructions are simply expanded to and or or handled appropriately.

4.5 Predefined function and predicate calls

We consider the following predefined function and predicate calls:
- $c \equiv sv([exp_1, \ldots, exp_n])$. The purpose of this Boolean function is to ensure that all the expressions correspond to standard values. Therefore: $\tau_s(c) = (\tau_e(exp_1) = z) \wedge \cdots \wedge (\tau_e(exp_n) = z)$, with z the value zero value associated to the type of the expressions.

- $c \equiv predef(f)(exp_1, \ldots, exp_n)$, or $c \equiv exp_1\ predef(f)\ exp_2$, with f a predefined function or an infix operator. The purpose of *predef* is to indicate that this call corresponds to the predefined MiniZinc function/operator f even if it has been redefined by the user. Therefore: $\tau_s(c) = f(\tau_s(exp_1), \ldots, \tau_s(exp_n))$, or $\tau_s(c) = \tau_s(exp_1)\ f\ \tau_s(exp_2)$ if f is an infix operator, and $\tau_e(c) = z$. Thus, the user should ensure, usually by adding some constraints using *sv* that $exp_1, \ldots, exp_n$ 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: $\tau_s(c) = (\tau_s(exp_1) = \tau_s(exp_2) \wedge \tau_e(exp_1) = \tau_e(exp_2))$ and $\tau_e(c) = z$. 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$^+$ $\mathcal{M}$, denoted by $\tau(\mathcal{M})$ is obtained transforming each of this 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, $o(t) \in \{int, float, bool\}$	
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 constraint. 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

$$\textit{extended } t = [c_{-n}, \ldots, c_{-1}] ++ \text{type} ++ [c_1, \ldots, c_m]$$

creates an array of string *tnames*

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

and replaces each show(e) by

```
if(fix(τ_e(e))==0) then show(τ_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 ];
```

6 Extended Optimization Models

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.[5] Given a and b can be determined we transform minimize/maximize y to minimize/maximize $\tau_e(y) * (b - a + 1) + \tau_s(y)$.

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

```
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 (c /\ x + y>23 /\ r=oneDayOrMore)
6                 \/ (c /\ x + y<=23 /\ r=x+y ) \/
7                 (not c /\ r=oneDayOrMore) } in r;
8 time: t1 = 5;
9 var time:t2;
10 var time:total = t1 + t2 + 21;
11 solve minimize total;
12 output(["Total=",show(total)," t2=",show(t2),"\n"]);
```

Fig. 3: Modelling time with an extended value..

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 $b = 23$.

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
```

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 σ a substitution. We say that σ is a* well-typed *substitution for $\mathcal{M}$ iff*

- *The domain of σ is the set containing the decision variables declared in $\mathcal{M}$.*[6]
- *For all $x \in dom(\sigma)$, $\Gamma \vdash x ::< t >$ iff $\Gamma \vdash x\sigma ::< t >$*

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.

[6] 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.

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

1. $\| id \|_\sigma = id\sigma$, id *any identifier.*
2. $\| k \|_\sigma = k$, k *any constant.*
3. *Set Expressions:*
   (a) $\| \{e_1, \ldots, e_n\} \|_\sigma = ord(\{\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma\})$.
   ord is defined as the function that given a set of values, eliminate the repetitions and sort the values according to order $\prec$ that extends ord_t defined in Section 4.1 where:
   $$a \prec b = \begin{cases} a < b & a, b \text{ standard constants} \\ ord_t(a) < ord_t(b) & \text{otherwise} \end{cases}$$
   (b) $\| e_i..e_f \|_\sigma = \{\| e_i \|_\sigma, \| e_i \|_\sigma + 1, \ldots, \| e_f \|_\sigma\}$
4. *Array Expressions:* $\| [e_1, \ldots, e_n] \|_\sigma = [\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma]$
5. *Array Access:*
   (a) $\| a[e] \|_\sigma = \| a \|_\sigma [\| e - (m-1) \|_\sigma]$, *$a$ an array identifier with index range $m \ldots n$, and $\| e \|_\sigma$ and integer value such that $m \leq \| e \|_\sigma \leq n$.*
   (b) $\| e_1[e_2] \|_\sigma = \| e_1 \|_\sigma [(\| e_2 \|_\sigma)]$, *$e_1$ not an array identifier, $\| e_1 \|_\sigma$ an array literal of n elements, and $\| e_2 \|_\sigma$ and integer value such that $1 \leq \| e_2 \|_\sigma \leq n$.*
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*
   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, $\| lc \|_\sigma$ is defined recursively as:
   (a) *If $m = 1$, then lc contains only one generator g, which must be of the form id in e'. Let $\| e' \|_\sigma$ be $\langle e_1, \ldots, e_n \rangle$ then:*
   $\| \langle e \mid g \text{ where } c\rangle \|_\sigma =$
   $\mathcal{C}(\| e \|_{\sigma \uplus \{id \mapsto e_1\}}, \| c \|_{\sigma \uplus \{id \mapsto e_1\}}) \ \diamond \ldots \diamond \ \mathcal{C}(\| e \|_{\sigma \uplus \{id \mapsto e_n\}}, \| c \|_{\sigma \uplus \{id \mapsto e_n\}})$
   (b) *If $m > 1$ then lc contains more than one generator. Analogously to the previous item, suppose that the first generator is of the form id in e', and let $\| e' \|_\sigma$ be $\langle e_1, \ldots, e_n \rangle$ then:*

   $\| \langle e \mid g_1, \ldots, g_m \text{ where } c\rangle \|_\sigma =$
   $\| \langle e \mid g_2 \ldots, g_m \text{ where } c\rangle \|_{\sigma \uplus \{id \mapsto e_1\}} \diamond \ldots \diamond \| \langle e \mid g_2 \ldots, g_m \text{ where } c\rangle \|_{\sigma \uplus \{id \mapsto e_n\}}$
7. $\| sv([e_1, \ldots, e_n]) \|_\sigma = st(t_1) \wedge \cdots \wedge st(t_n)$ *with $\Gamma \vdash \| e_1 \|_\sigma :: t_1, \Gamma \vdash \| e_n \|_\sigma :: t_n$*
8. $\| e_1 = e_2 \|_\sigma = true$ *if $\| e_1 \|_\sigma$ and $\| e_2 \|_\sigma$ are the same constant, false otherwise.*

9. $\| p(e_1, \ldots, e_n) \|_\sigma = p(\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma)$, *with* p MiniZinc *predefined (that* p *is a relational operator or predefined arithmetic function such as* $>, <, +$ $\ldots$*)* .
10. Forall, exists *constructions:*
    Let $\| a \|_\sigma$ *be* $[v_1, \ldots, v_n]$*, then:*
    – $\| forall(a) \|_\sigma = v_1 \wedge \cdots \wedge v_n$
    – $\| exists(a) \|_\sigma = v_1 \vee \cdots \vee v_n$

Thus, the overall idea of the evaluation is simply to evaluate the expressions after replacing the variables by their values. The next points describe in detail the more involved parts of this definition:

– In the evaluation of a set literal $\{e_1, \ldots, e_n\}$, item 3a, the result is a set where the elements are enumerated following an order. Although this is not important because the resulting set is the same and could be omitted it has been included to represent the semantics of the actual MiniZinc systems. For instance, in standard MiniZinc:

```
var set of 1..10: x;
constraint x={5,4,3,2,1};
solve satisfy;
output([show(x)]);
```

  displays the answer *1..5*, and

```
var int: x;
constraint x=[i | i in {9,8,7,6,5}][2];
solve satisfy;
output([show(x)]);
```

  displays *6*.
– In the array access of the form $a[e]$ with a an identifier, item 5a. We first obtain the assignment for a, which must be in the substitution σ as an array literal. Then e is evaluated to obtain the index. Finally, the array is accessed *after* correcting the index value. The correction is needed because array literals always start with index 1.
– The evaluation of list/set comprehensions is a little bit more involved, but corresponds closely to the intuition about how this structures should be evaluated. In the case of only one generator we obtain the list/set with all the expressions that it can generate, remove those that do not satisfy the guard c (this is done by $\mathcal{C}(e, c)$), and replace the variable associated to the generator by the corresponding values, combining all the results in a single list/set (operator $\diamond$). If there are two or more generators we follow the same approach for removing the first generator and proceed recursively.

Now we can define the concept of solution.

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

1. *For every assignment* a *in* A*,* $\| a \|_\sigma = true$*.*

2. *For every constraint* constraint c *in* C, $\| c \|_\sigma = true$.
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* $f\sigma > f\sigma$ *(respectively* $f\sigma < f\sigma$*)*

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

$$\sigma^{\mathcal{T}} = \{\tau_s(x) \mapsto \tau_s(v) \mid (x \mapsto v) \in \sigma\} \cup \\ \{\tau_e(x) \mapsto \tau_e(v) \mid (x \mapsto v) \in \sigma, \tau_e(x) \neq z\}$$

We first introduce some useful technical results.

The first one indicates that after applying t_s, t_e to constants no further evaluation is needed.

Lemma 1. *If* e *is a MiniZinc*$^+$ *constant then* $\| \tau_s(e) \|_\emptyset = \tau_s(e)$ *and* $\| \tau_e(e) \|_\emptyset = \tau_e(e)$.

Proof. If e is a constant then both $\tau_s(e)$ and $\tau_e(e)$ are constants and Definition 22 shows that the evaluation behaves as the identity on constants.

Lemma 2. *Let* $\circ$ *be a* MiniZinc *relational operator,* k *a constant,* $\diamond$, $\langle , \rangle$ *and* $\mathcal{C}(e, c)$ *as in definition 2.6 then:*

$$\|\langle [a_1, \ldots, a_n][i] \mid i \textit{ in } 1..n \textit{ where } [b_1, \ldots, b_n][i] \circ k \rangle\|_\sigma = \\ \mathcal{C}(\| a_1 \|_\sigma, \| b_1 \|_\sigma \circ k) \diamond \ldots \diamond \mathcal{C}(\| a_n \|_\sigma, \| b_n \|_\sigma \circ k)$$

Proof. In the expression

$$\|\langle [a_1, \ldots, a_n][i] \mid i \text{ in } 1..n \text{ where } [b_1, \ldots, b_n][i] \circ k \rangle\|_\sigma$$

there is only one generator, and therefore the evaluation is defined in 2.6a.

$$\begin{aligned} &\|\langle [a_1, \ldots, a_n][i] \mid i \text{ in } 1..n \text{ where} \\ &\qquad [b_1, \ldots, b_n][i] \circ k \rangle\|_\emptyset = \\ &\qquad \text{By Definition 2.6a} \\ &\mathcal{C}(\| [a_1, \ldots, a_n][i] \|_{\sigma \uplus i \mapsto 1}, \| [b_1, \ldots, b_n][i] \circ k \|_{\sigma \uplus i \mapsto 1}) \diamond \ldots \\ &\quad \diamond \mathcal{C}(\| [a_1, \ldots, a_n][i] \|_{\sigma \uplus i \mapsto n}, \| [b_1, \ldots, b_n][i] \circ k \|_{\sigma \uplus i \mapsto n}) = \\ &\qquad \text{By Definition 2.9} \end{aligned}$$

$$\begin{array}{r}
\mathcal{C}(\| [a_1,\ldots,a_n][i] \|_{\sigma\uplus i\mapsto 1}, \| [b_1,\ldots,b_n][i] \|_{\sigma\uplus i\mapsto n} \circ \| k \|_{\sigma\uplus i\mapsto 1}) \diamond \ldots \\
\diamond\mathcal{C}(\| [a_1,\ldots,a_n][i] \|_{\sigma\uplus i\mapsto n}, \| [b_1,\ldots,b_n][i] \|_{\sigma\uplus i\mapsto n} \circ \| k \|_{\sigma\uplus i\mapsto n}) = \\
\text{By definition 2.5b} \\
\mathcal{C}([\| a_1 \|_{\sigma\uplus i\mapsto 1},\ldots,\| a_n \|_{\sigma\uplus i\mapsto 1}][\| i \|_{\sigma\uplus i\mapsto 1}], \\
[\| b_1 \|_{\sigma\uplus i\mapsto 1},\ldots,\| b_n \|_{\sigma\uplus i\mapsto 1}][\| i \|_{\sigma\uplus i\mapsto 1}] \circ \| k \|_{\sigma\uplus i\mapsto 1}) \diamond \ldots \\
\diamond\mathcal{C}([\| a_1 \|_{\sigma\uplus i\mapsto n},\ldots,\| a_n \|_{\sigma\uplus i\mapsto n}][\| i \|_{\sigma} \uplus i \mapsto n], \\
[\| b_1 \|_{\sigma\uplus i\mapsto n},\ldots,\| b_n \|_{\sigma\uplus i\mapsto n}][\| i \|_{\sigma\uplus i\mapsto n}] \circ \| k \|_{\sigma\uplus i\mapsto 1}) = \\
\text{By Definitions 2.1 and 2.2, having} \\
i \text{ does not appear in } [a_1,\ldots,a_n] \text{ or } [b_1,\ldots,b_n] \\
\mathcal{C}([\| a_1 \|_{\sigma},\ldots,\| a_n \|_{\sigma}][1], [\| b_1 \|_{\sigma},\ldots,\| b_n \|_{\sigma}][1] \circ k) \diamond \ldots \\
\diamond\mathcal{C}([\| a_1 \|_{\sigma},\ldots,\| a_n \|_{\sigma}][n], [\| b_1 \|_{\sigma},\ldots,\| b_n \|_{\sigma}][n] \circ k) = \\
\text{By array access} \\
\mathcal{C}(\| a_1 \|_{\sigma}, \| b_1 \|_{\sigma} \circ k) \diamond \ldots \mathcal{C}(\| a_n \|_{\sigma}, \| b_n \|_{\sigma} \circ k)
\end{array}$$

The soundness of the proposal will be a direct consequence of the next auxiliary result:

Lemma 3. *For every expression e and well-typed substitution σ:*

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

where $\emptyset$ denotes the identity substitution.

Proof. Structural induction on the form of e:

- e is an identifier. Then:

$$\text{There is some } v \text{ such that } e\sigma = v \text{ (Def. 1)} \quad (1)$$

Moreover, by Definition 4

$$e \mapsto v \in \sigma \text{ iff } \tau_s(e) \mapsto \tau_s(v) \in \sigma^{\mathcal{T}} \quad (2)$$

$$e \mapsto v \in \sigma, \tau_e(e) \neq z \text{ iff } \tau_e(e) \mapsto \tau_e(v) \in \sigma^{\mathcal{T}} \quad (3)$$

We distinguish two cases:

- If e is an standard type identifier.

$$\begin{array}{l}
\| \tau_s(\| e \|_{\sigma}) \|_{\emptyset} = \text{ (Def. 2.1)} = \| \tau_s(e\sigma) \|_{\emptyset} = \text{ (Lemma 1)} = \\
\qquad = \tau_s(e\sigma) = \text{(By (1))} = \underline{\tau_s(v)} \\
\| \tau_s(e) \|_{\sigma^{\mathcal{T}}} = \text{(By (2))} = \underline{\tau_s(v)}
\end{array}$$

Thus, $\| \tau_s(\| e \|_{\sigma}) \|_{\emptyset} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}}$. In order to check that $\| \tau_e(\| e \|_{\sigma}) \|_{\emptyset} = \| \tau_e(e) \|_{\sigma^{\mathcal{T}}}$ observe that since e is of an standard type t then $\tau_e(e) = z_t$, and

since σ is well-typed (Def. 1) then $\| e \|_\sigma = e\sigma$ is a constant of type t, and therefore $\tau_e(e\sigma) = z_t$

$$\begin{array}{rl} \| \tau_e(\| e \|_\sigma) \|_\emptyset = & (\text{Def. } 2.1) = \| \tau_e(e\sigma) \|_\emptyset = \; e\sigma \text{ of type } t = \\ & = \| z_t \|_\emptyset = \; (\text{Lemma } 1) = \underline{z_t} \\ \| \tau_e(e) \|_{\sigma^\mathcal{T}} = & (e \text{ standard}) = \| z_t \|_{\sigma^\mathcal{T}} = \; (\text{Def. } 2.2) = \underline{z_t} \end{array}$$

• If e is an extended type identifier, then the proof for τ_s in the previous case is here valid for both tau_s and τ_e. Let t represent either s or e. Then:

$$\begin{array}{rl} \| \tau_t(\| e \|_\sigma) \|_\emptyset = & (\text{Def. } 2.1) = \| \tau_t(e\sigma) \|_\emptyset = \; (\text{Lemma } 1) = \\ & = \tau_t(e\sigma) = (\text{By } (1)) = \underline{\tau_t(v)} \\ \| \tau_s(e) \|_{\sigma^\mathcal{T}} = & (\text{Either by } (2) \text{ if } t \text{ is } s \text{ or by } (3) \text{ if } t \text{ is } e) = \underline{\tau_t(v)} \end{array}$$

Thus, $\| \tau_t(\| e \|_\sigma) \|_\emptyset = \| \tau_t(e) \|_{\sigma^\mathcal{T}}$.

– e is a base type constant:
Depending on its type:
  • e is an standard base type constant:

$$\begin{aligned} \| \tau_s(\| e \|_\sigma) \|_\emptyset &= \text{By Definition 2.2} \\ \| \tau_s(e) \|_\emptyset &= \text{By constant transformation} \\ \| e \|_\emptyset &= \text{By Definition 2.2} \\ e &= \text{By Definition 2.2} \\ \| e \|_{\sigma^\mathcal{T}} &= \text{By constant transformation} \\ \| \tau_s(e) \|_{\sigma^\mathcal{T}} & \end{aligned}$$

$$\begin{aligned} \| \tau_e(\| e \|_\sigma) \|_\emptyset &= \text{By Definition 2.2} \\ \| \tau_e(e) \|_\emptyset &= \text{By constant transformation} \\ \| 0 \|_\emptyset &= \text{By Definition 2.2} \\ 0 &= \text{By Definition 2.2} \\ \| 0 \|_{\sigma^\mathcal{T}} &= \text{By constant transformation} \\ \| \tau_e(e) \|_{\sigma^\mathcal{T}} & \end{aligned}$$

  • e is an extended base type constant:

$$\begin{aligned} \| \tau_s(\| e \|_\sigma) \|_\emptyset &= \text{By Definition 2.2} \\ \| \tau_s(e) \|_\emptyset &= \text{By constant transformation} \\ \| k_{o(t)} \|_\emptyset &= \text{By Definition 2.2} \\ k_{o(t)} &= \text{By Definition 2.2} \\ \| k_{o(t)} \|_{\sigma^\mathcal{T}} &= \text{By constant transformation} \\ \| \tau_s(e) \|_{\sigma^\mathcal{T}} & \end{aligned}$$

$$\begin{aligned}
\|\,\tau_e(\|\,e\,\|_\sigma)\,\|_\emptyset &= \text{By Definition 2.2}\\
\|\,\tau_e(e)\,\|_\emptyset &= \text{By constant transformation}\\
\|\,ord_t(k)\,\|_\emptyset &= \text{By Definition 2.2}\\
ord_t(k) &= \text{By Definition 2.2}\\
\|\,ord_t(k)\,\|_{\sigma^\mathcal{T}} &= \text{By constant transformation}\\
\|\,\tau_e(e)\,\|_{\sigma^\mathcal{T}}
\end{aligned}$$

– $e \equiv a[i]$ is an array access, with a an array identifier with index range $m \ldots n$, and $\|\,i\,\|_\sigma$ and integer expression such that $m \leq \|\,i\,\|_\sigma \leq n$.
By Def. 5a, $\|\,a[i]\,\|_\sigma = \|\,a\,\|_\sigma[\|\,i-(m-1)\,\|_\sigma]$. Let t be either s or e.

$$\begin{aligned}
\|\,\tau_t(\|\,a[i]\,\|_\sigma)\,\|_\emptyset &= \text{By Definition 2.5a}\\
\|\,\tau_t(\|\,a\,\|_\sigma[\|\,i-(m-1)\,\|_\sigma])\,\|_\emptyset &= \text{By array access transformation}\\
\|\,\tau_t(\|\,a\,\|_\sigma)[\tau_s(\|\,i-(m-1)\,\|_\sigma)]\,\|_\emptyset &= \text{By Definition 2.5a}\\
\|\,\tau_t(\|\,a\,\|_\sigma)\,\|_\emptyset[\|\,\tau_s(\|\,e\,\|_\sigma)\,\|_\emptyset] &= \text{By induction hypotheses}\\
\|\,\tau_t(a)\,\|_{\sigma^\mathcal{T}}[\|\,\tau_s(i-(m-1))\,\|_{\sigma^\mathcal{T}}] &= \text{By Definition 2.5a}\\
\|\,\tau_t(a)[\tau_s(i)]\,\|_{\sigma^\mathcal{T}} &= \text{By array access transformation}\\
\|\,\tau_t(a[i])\,\|_{\sigma^\mathcal{T}}
\end{aligned}$$

– $e \equiv e'[i]$ is an array access, with e' an array expression and i an integer expression. Analogous to the previous case.
– e is a set expression

Let

$$ord(\{\|\,e_1\,\|_\sigma, \ldots, \|\,e_n\,\|_\sigma\}) = \{\|\,e_{p_1}\,\|_\sigma, \ldots, \|\,e_{p_m}\,\|_\sigma\} \tag{4}$$

- If e is a set of standard type of the form $\{e_1, \ldots, e_n\}$ then $\tau_s(e) = \{\tau_s(e_1), \ldots, \tau_s(e_n)\}$ and $\tau_e(e) = \{\}$

$$\begin{aligned}
\|\,\tau_s(\|\,\{e_1, \ldots, e_n\}\,\|_\sigma)\,\|_\emptyset &= &&\text{By Definition 2.3a}\\
\|\,\tau_s(\{ord(\|\,e_1\,\|_\sigma, \ldots, \|\,e_n\,\|_\sigma)\})\,\|_\emptyset &= &&\text{By (4)}\\
\|\,\tau_s(\{\|\,e_{p_1}\,\|_\sigma, \ldots, \|\,e_{p_m}\,\|_\sigma\})\,\|_\emptyset &= &&\text{By set transformation}\\
\|\,\{\tau_s(\|\,e_{p_1}\,\|_\sigma), \ldots, \tau_s(\|\,e_{p_m}\,\|_\sigma)\}\,\|_\emptyset &= &&\text{By Definition 2.3a}\\
ord(\{\|\,\tau_s(\|\,e_{p_1}\,\|_\sigma)\,\|_\emptyset, \ldots, \|\,\tau_s(\|\,e_{p_m}\,\|_\sigma)\,\|_\emptyset\}) & &&(5)
\end{aligned}$$

$$\begin{aligned}
\|\,\tau_s(\{e_1, \ldots, e_n\})\,\|_{\sigma^\mathcal{T}} &= &&\text{By set transformation}\\
\|\,\{\tau_s(e_1), \ldots, \tau_s(e_n)\}\,\|_{\sigma^\mathcal{T}} &= &&\text{By Definition 2.3a}\\
ord(\{\|\,\tau_s(e_1)\,\|_{\sigma^\mathcal{T}}, \ldots, \|\,\tau_s(e_n)\,\|_{\sigma^\mathcal{T}}\}) &= &&\text{By Lemma 3}\\
ord(\{\|\,\tau_s(\|\,e_1\,\|_\sigma)\,\|_\emptyset, \ldots, \|\,\tau_s(\|\,e_n\,\|_\sigma)\,\|_\emptyset\}) &= &&\text{By (4), repetition elimination and reordering}\\
ord(\{\|\,\tau_s(\|\,e_{p_1}\,\|_\sigma)\,\|_\emptyset, \ldots, \|\,\tau_s(\|\,e_{p_m}\,\|_\sigma)\,\|_\emptyset\}) & &&(6)
\end{aligned}$$

By (5) = (6), $\parallel \tau_s(\parallel \{e_1, \ldots, e_n\} \parallel_\sigma) \parallel_\emptyset = \parallel \tau_s(\{e_1, \ldots, e_n\}) \parallel_{\sigma^\mathcal{T}}$

$$
\begin{aligned}
\parallel \tau_e(\parallel e \parallel_\sigma) \parallel_\emptyset = & \quad \text{By Definition 2.3a} \\
\parallel \tau_e(ord(\{\parallel e_1 \parallel_\sigma, \ldots, \parallel e_n \parallel_\sigma\})) \parallel_\emptyset = & \quad \text{By set transformation} \\
\parallel ord(\{\}) \parallel_\emptyset = & \\
\parallel \{\} \parallel_\emptyset = & \quad \text{By Definition 2.3a} \\
ord(\{\}) = & \quad \text{By Definition 2.3a} \\
\parallel \{\} \parallel_{\sigma^\mathcal{T}} = & \quad \text{By set transformation} \\
\parallel \tau_e(e) \parallel_{\sigma^\mathcal{T}} &
\end{aligned}
$$

- If e is a set of extended type of the form $\{e_1, \ldots, e_n\}$ then:

$$
\begin{aligned}
& \parallel \tau_s(\parallel e \parallel_\sigma) \parallel_\emptyset = \\
& \parallel \tau_s(ord(\{\parallel e_1 \parallel_\sigma, \ldots, \parallel e_n \parallel_\sigma\})) \parallel_\emptyset = \\
& \text{By (4)} \\
& \parallel \tau_s(\{\parallel e_{p_1} \parallel_\sigma, \ldots, \parallel e_{p_m} \parallel_\sigma\}) \parallel_\emptyset = \\
& \text{By set transformation} \\
& \parallel \{[\tau_s(\parallel e_{p_1} \parallel_\sigma), \ldots, \tau_s(\parallel e_{p_m} \parallel_\sigma)][i] \;\mid\; i \text{ in } 1..m \\
& \text{where } [\tau_e(\parallel e_{p_1} \parallel_\sigma), \ldots, \tau_e(\parallel e_{p_m} \parallel_\sigma)][i] = 0\} \parallel_\emptyset = \\
& \text{By (2)} \\
& \mathcal{C}(\parallel \tau_s(\parallel e_{p_1} \parallel_\sigma) \parallel_\emptyset, \parallel \tau_e(\parallel e_{p_1} \parallel_\sigma) \parallel_\emptyset = 0) \text{ union} \\
& \ldots \text{ union } \mathcal{C}(\parallel \tau_s(\parallel e_{p_m} \parallel_\sigma) \parallel_\emptyset, \parallel \tau_e(\parallel e_{p_m} \parallel_\sigma) \parallel_\emptyset = 0) = \\
& \text{By I.H.} \\
& \mathcal{C}(\parallel \tau_s(e_{p_1}) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_{p_1}) \parallel_{\sigma^\mathcal{T}} = 0) \text{ union} \qquad (7) \\
& \ldots \text{ union } \mathcal{C}(\parallel \tau_s(e_{p_m}) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_{p_m}) \parallel_{\sigma^\mathcal{T}} = 0)
\end{aligned}
$$

$$
\begin{aligned}
& \parallel \tau_s(e) \parallel_{\sigma^\mathcal{T}} = \\
& \text{By set transformation} \\
& \parallel \{[\tau_s(e_1), \ldots, \tau_s(e_n)][i] \;\mid\; i \; in \; 1..n \\
& \text{where } [\tau_e(e_1), \ldots, \tau_e(e_n)][i] = 0\} \parallel_{\sigma^\mathcal{T}} = \\
& \text{By (2)} \\
& \mathcal{C}(\parallel \tau_s(e_1) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_1) = 0 \parallel_{\sigma^\mathcal{T}}) \text{ union} \\
& \ldots \text{union } \mathcal{C}(\parallel \tau_s(e_n) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_n) = 0 \parallel_{\sigma^\mathcal{T}}) = \qquad (8) \\
& \text{By (4), repetition elimination and reordering} \\
& \mathcal{C}(\parallel \tau_s(e_{p_1}) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_{p_1}) = 0 \parallel_{\sigma^\mathcal{T}}) \text{ union} \qquad (9) \\
& \ldots \text{union } \mathcal{C}(\parallel \tau_s(e_{p_m}) \parallel_{\sigma^\mathcal{T}}, \parallel \tau_e(e_{p_m}) = 0 \parallel_{\sigma^\mathcal{T}})
\end{aligned}
$$

By (7) = (9), $\parallel \tau_s(\parallel e \parallel_\sigma) \parallel_\emptyset = \parallel \tau_s(e) \parallel_{\sigma^\mathcal{T}}$.
Notice that *union* is the set union and therefore step (8) can be performed.

– e is an array expression of the form $[e_1, \ldots, e_n]$:
Let t be either s or e, then:

$$\begin{aligned}
\| \tau_t(\| e \|_\sigma) \|_\emptyset &= \\
\| \tau_t(\| [e_1, \ldots, e_n] \|_\sigma) \|_\emptyset &= \text{By Definition 2.4} \\
\| \tau_t([\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma]) \|_\emptyset &= \text{By array transformation} \\
\| [\tau_t(\| e_1 \|_\sigma), \ldots, \tau_t(\| e_n \|_\sigma)] \|_\emptyset &= \text{By Definition 2.4} \\
[\| \tau_t(\| e_1 \|_\sigma) \|_\emptyset, \ldots, \| \tau_t(\| e_n \|_\sigma) \|_\emptyset] &= \text{By I.H.} \\
\| \tau_t(e_1) \|_{\sigma\tau}, \ldots, \| \tau_t(e_n) \|_{\sigma\tau} &= \text{By Definition 2.4} \\
\| [\tau_t(e_1), \ldots, \tau_t(e_n)] \|_{\sigma\tau} &= \text{By array transformation} \\
\| \tau_t([e_1, \ldots, e_n]) \|_{\sigma\tau} &= \\
\| \tau_t(e) \|_{\sigma\tau}
\end{aligned}$$

– e is a set/list comprehension of the form $\langle exp \mid G_1, \ldots, G_m \text{ where } cond \rangle$, with $\langle , \rangle$ representing either [,] or {,}.

- If e has only one generator G_m of the form g_m in ge_m with $\| ge_m \|_\sigma = \langle e_1, \ldots, e_n \rangle$:
  Let $\tau_g(c)$ be:
  - $\tau_s(c)$ if e is a list comprehension
  - $\tau_s(c) \wedge \tau_e(exp) = 0$ if e is a set comprehension

  Let t be either s or e. By comprehension expression transformation $\tau_t(\langle exp \mid G_m \text{ where } cond \rangle) = \| \langle \tau_t(exp') \mid G'_m \text{ where } \tau_g(cond') \rangle \|_{\sigma\tau}$ where:
  - If g_m is a set or array of standard type then $exp' = exp$, $cond' = cond$ and $G'_m = g_m$ in $\tau_s(ge_m)$
  - If g_m is a set or array of extended type then:
    - Array a is defined as ge_m if ge_m is an array expression and as $[ord_t^{-1}(\mathsf{x}) \mid \mathsf{x} \text{ in } \tau_e(ge_m) \text{ where } \mathsf{x}{<}0]{+}{+}[\mathsf{x} \mid \mathsf{x} \text{ in } \tau_s(ge_m)]{+}{+}[ord_t^{-1}(\mathsf{x}) \mid \mathsf{x} \text{ in } \tau_e(ge_m) \text{ where } \mathsf{x}{>}0]$ if ge_m is a set expression.
    - $G'_m = f$ in index-set(a) with f a free variable.
    - exp' and $cond'$ are exp and $cond$ where each ocurrence of g_m has been changed by the array access $a[f]$.

  First lets see that if $\| ge_m \|_\sigma = \langle e_1, \ldots, e_m \rangle$ then $\| a \|_\sigma = [e_1, \ldots, e_n]$:

  - If ge_m is an array expression $a = ge_m$ and therefore $\| ge_m \|_\sigma = \langle e_1, \ldots, e_m \rangle$ iff $\| a \|_\sigma = [e_1, \ldots, e_n]$
  - If ge_m is a set expression: By set evaluation definition (Definition 2.3a) following statements holds:

$$e_1 \prec e_2 \prec \ldots \prec e_n \tag{10}$$

$$
\begin{aligned}
\exists i, j \in 1..n \mid i \leq j \wedge & \\
& (\forall k \in 1..i-1\ \tau_e(e_k) < 0 \wedge \\
& \forall k \in i..j-1\ \tau_e(e_k) = 0 \wedge \\
& \forall k \in i..n\ \tau_e(e_k) > 0) && (11)\\
& \wedge\ \mathit{ord}(\tau_e(\{\tau_e(e_1), \ldots, \tau_e(e_{i-1})\})) = \\
& \{\tau_e(e_1), \ldots, \tau_e(e_{i-1})\} && (12)\\
& \wedge\ \mathit{ord}(\tau_e(\{\tau_s(e_i), \ldots, \tau_s(e_{j-1})\})) = \\
& \{\tau_s(e_i), \ldots, \tau_s(e_{j-1})\} && (13)\\
& \wedge\ \mathit{ord}(\tau_e(\{\tau_s(e_j), \ldots, \tau_s(e_n)\})) = \\
& \{\tau_e(e_j), \ldots, \tau_e(e_n)\} && (14)
\end{aligned}
$$

We must prove :
$\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x < 0] \|_{\sigma\tau} = [e_1, \ldots, e_{i-1}]$,
$\| [x | x \text{ in } \tau_s(ge_m)] \|_{\sigma\tau} = [e_i, \ldots, e_{j-1}]$ and
$\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x > 0] \|_{\sigma\tau} = [e_j, \ldots, e_n]$

To evaluate these expressions we have to calculate $\| \tau_s(ge_m) \|_{\sigma\tau}$ and $\| \tau_e(ge_m) \|_{\sigma\tau}$:

$$
\| \tau_s(ge_m) \|_{\sigma\tau} = \text{By Lemma 3} \qquad (15)
$$

$$
\begin{aligned}
\| \tau_s(\| ge_m \|_\sigma) \|_\emptyset &= \\
\| \tau_s(\{e_1, \ldots, e_m\}) \|_\emptyset &= \text{By set transformation} \\
\| \{[\tau_s(e_1), \ldots, \tau_s(e_n)][i] \mid i \text{ in } 1..n \text{ where} & \\
[\tau_e(e_1), \ldots, \tau_e(e_n)][i] = 0\} \|_\emptyset &= \text{By (2)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}(\| \tau_s(e_1) \|_\emptyset, \| \tau_e(e_1) \|_\emptyset = 0) \text{ union } \ldots & \\
\text{union } \mathcal{C}(\| \tau_s(e_n) \|_\emptyset, \| \tau_e(e_n) \|_\emptyset = 0) &= \\
\text{By (11) and } \mathcal{C}(,) \text{ definition} & \\
\{\| \tau_s(e_i) \|_\emptyset\} \text{ union } \ldots \text{ union } \{\| \tau_s(e_{j-1}) \|_\emptyset\} &= \\
\text{By Definition 2.3a} & \\
\| \{\tau_s(e_i)\} \|_\emptyset \text{ union } \ldots \text{ union } \| \{\tau_s(e_{j-1})\} \|_\emptyset &= \\
\text{By Definition 2.9} &
\end{aligned}
$$

$$\| \{\tau_s(e_i)\} \text{ union } \dots \text{ union } \{\tau_s(e_{j-1})\} \|_\emptyset =$$

Having by (11) and (13) that the sets are disjunct

$$\| \{\tau_s(e_i), \dots, \tau_s(e_{j-1})\} \|_\emptyset =$$

By Definition 3a

$$ord(\{\| \tau_s(e_i) \|_\emptyset, \dots, \| \tau_s(e_i) \|_\emptyset\}) =$$

By identity

$$ord(\{\tau_s(e_i), \dots, \tau_s(e_{j-1})\}) =$$

By (13)

$$\{\tau_s(e_i), \dots, \tau_s(e_{j-1})\}$$

$$\| \tau_e(ge_m) \|_\sigma \tau =$$

By Lemma 3 (16)

$$\| \tau_e(\| ge_m \|_\sigma) \|_\emptyset =$$

$$\| \tau_e(\{e_1, \dots, e_m\}) \|_\emptyset =$$

By set transformation

$$\|\{[\tau_e(e_1), \dots, \tau_e(e_n)][i] \mid i \text{ in } 1..n \text{ where}$$

$$[\tau_e(e_1), \dots, \tau_e(e_n)][i]! = 0\}\|_\emptyset =$$

By (2)

$$\mathcal{C}(\| \tau_e(e_1) \|_\emptyset, \| \tau_e(e_1) \|_\emptyset! = 0) \text{ union } \dots$$

$$\text{union } \mathcal{C}(\| \tau_e(e_n) \|_\emptyset, \| \tau_e(e_n) \|_\emptyset! = 0)$$

$$= \text{By (11) and } \mathcal{C}(,) \text{ definition}$$

$$\{\| \tau_e(e_1) \|_\emptyset\} \text{ union } \dots \text{ union } \{\| \tau_e(e_{i-1}) \|_\emptyset\}$$

$$\text{union } \{\| \tau_e(e_j) \|_\emptyset \text{ union } \} \dots \text{ union } \{\| \tau_e(e_n) \|_\emptyset\} =$$

By Definition 2.3a

$$\| \{\tau_e(e_1)\} \|_\emptyset \text{ union } \dots \text{ union } \| \{\tau_e(e_{i-1})\} \|_\emptyset$$

$$\text{union } \| \{\tau_e(e_j)\} \|_\emptyset \text{ union } \dots \text{ union } \| \{\tau_e(e_n)\} \|_\emptyset =$$

By Definition 2.9

$$\|\{\tau_e(e_1)\} \text{ union } \dots \text{ union } \{\tau_e(e_{i-1})\}$$

$$\text{union } \{\tau_e(e_j)\} \text{ union } \dots \text{ union } \{\tau_e(e_n)\}\|_\emptyset =$$

Having by (11), (12) and (14) that the sets are disjunct

$$\| \{\tau_e(e_1), \dots, \tau_e(e_{i-1}), \tau_e(e_j), \dots, \tau_e(e_n)\} \|_\emptyset$$

$$
\begin{aligned}
&\text{By Definition 2.3a}\\
ord(\{\| \tau_e(e_1) \|_\emptyset, \ldots, \| \tau_e(e_{i-1}) \|_\emptyset, \| \tau_e(e_j) \|_\emptyset, \ldots, \| \tau_e(e_n) \|_\emptyset\}) &=\\
&\text{By identity}\\
ord(\{\tau_e(e_1), \ldots, \tau_e(e_{i-1}), \tau_e(e_j), \ldots, \tau_e(e_n)\}) &=\\
&\text{By (11), (12) and (14)}\\
\{\tau_e(e_1), \ldots, \tau_e(e_{i-1}), \tau_e(e_j), \ldots, \tau_e(e_n)\}
\end{aligned}
$$

Lets see now that:
$\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x < 0] \|_{\sigma}\tau = [e_1, \ldots, e_{i-1}]$,
$\| [x | x \text{ in } \tau_s(ge_m)] \|_{\sigma}\tau = [e_i, \ldots, e_{j-1}]$ and
$\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x > 0] \|_{\sigma}\tau = [e_j, \ldots, e_n]$:

1. $\| [x | x \text{ in } \tau_s(ge_m)] \|_{\sigma}\tau = \| [e_i, \ldots, e_{j-1}] \|_{\sigma}\tau$:

$$
\begin{aligned}
\| [x | x \text{ in } \tau_s(ge_m)] \|_{\sigma}\tau &=\\
&\text{Is syntax sugar of:}\\
\| [x | x \text{ in } \tau_s(ge_m)] \text{ where } true \|_{\sigma}\tau &=\\
&\text{By previous } \tau_s(ge_m) \text{ evaluation and Definition 2.6a}\\
\mathcal{C}(\| x \|_{\sigma\tau \uplus x \mapsto \tau_s(e_i)}, \| true \|_{\sigma\tau \uplus x \mapsto \tau_s(e_i)}) &++\\
\cdots ++ \mathcal{C}(\| x \|_{\sigma\tau \uplus x \mapsto \tau_s(e_{j-1})}, \| true \|_{\sigma\tau \uplus x \mapsto \tau_s(e_{j-1})}) &=\\
&\text{By Definition 2.2 and } \mathcal{C}(,) \text{ definition}
\end{aligned}
$$

$$
\begin{aligned}
[\| \tau_s(e_i) \|_{\sigma}\tau] ++ \cdots ++ [\| \tau_s(e_{j-1}) \|_{\sigma}\tau] &=\\
&\text{Applying concatenation}\\
[\| \tau_s(e_i) \|_{\sigma}\tau, \ldots, \| \tau_s(e_{j-1}) \|_{\sigma}\tau] &=\\
&\text{By constant transformation, having}\\
&(\tau_e(e_i) = z_t, \ldots, \tau_e(e_{j-1}) = z_t \text{ by (11)})\\
[\| e_i \|_{\sigma}\tau, \ldots, \| e_{j-1} \|_{\sigma}\tau] &=\\
&\text{By Definition 2.4}\\
\| [e_i, \ldots, e_{j-1}] \|_{\sigma}\tau
\end{aligned} \tag{17}
$$

2. $\|\, [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x < 0] \,\|_{\sigma}\tau = \|\, [e_1, \ldots, e_{i-1}] \,\|_{\sigma}\tau$:

$$
\begin{aligned}
&\|\, [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x < 0] \,\|_{\sigma}\tau = \\
&\text{By previous } \tau_s(ge_m) \text{ evaluation and Definition 2.6a} \\
&\mathcal{C}(\|\, ord_t^{-1}(x) \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_1)}, \|\, x < 0 \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_1)}) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(x) \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_{i-1})}, \|\, x < 0 \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_{i-1})}) \\
&++ \\
&\mathcal{C}(\|\, ord_t^{-1}(x) \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_j)}, \|\, x < 0 \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_j)}) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(x) \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_n)}, \|\, x < 0 \,\|_{\sigma}\tau_{\uplus x \mapsto \tau_e(e_n)}) = \\
&\text{By Definitions 2.2, 2.9} \\
&\mathcal{C}(\|\, ord_t^{-1}(\tau_e(e_1)) \,\|_{\sigma}\tau, \|\, \tau_e(e_1) \,\|_{\sigma}\tau < 0) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(\tau_e(e_{i-1})) \,\|_{\sigma}\tau, \|\, \tau_e(e_{i-1}) \,\|_{\sigma}\tau < 0) \\
&++ \\
&\mathcal{C}(\|\, ord_t^{-1}(\tau_e(e_j)) \,\|_{\sigma}\tau, \|\, \tau_e(e_j) \,\|_{\sigma}\tau < 0) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(\tau_e(e_n)) \,\|_{\sigma}\tau, \|\, \tau_e(e_n) \,\|_{\sigma}\tau < 0) = \\
&\text{By constant transformation} \\
&\mathcal{C}(\|\, ord_t^{-1}(ord_t(e_1)) \,\|_{\sigma}\tau, \|\, \tau_e(e_1) \,\|_{\sigma}\tau < 0) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(ord_t(e_{i-1})) \,\|_{\sigma}\tau, \|\, \tau_e(e_{i-1}) \,\|_{\sigma}\tau < 0) \\
&++ \\
&\mathcal{C}(\|\, ord_t^{-1}(ord_t(e_j)) \,\|_{\sigma}\tau, \|\, \tau_e(e_j) \,\|_{\sigma}\tau < 0) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, ord_t^{-1}(ord_t(e_n)) \,\|_{\sigma}\tau, \|\, \tau_e(e_n) \,\|_{\sigma}\tau < 0) = \\
&\mathcal{C}(\|\, e_1 \,\|_{\sigma}\tau, \|\, \tau_e(e_1) \,\|_{\sigma}\tau < 0) \\
&++ \cdots ++ \\
&\mathcal{C}(\|\, e_{i-1} \,\|_{\sigma}\tau, \|\, \tau_e(e_{i-1}) \,\|_{\sigma}\tau < 0) \\
&++
\end{aligned}
\tag{18}
$$

$$
\begin{aligned}
&\mathcal{C}(\| e_j \|_{\sigma\tau}, \| \tau_e(e_j) \|_{\sigma\tau} < 0) \\
&+ + \cdots + + \\
&\mathcal{C}(\| e_n \|_{\sigma\tau}, \| \tau_e(e_n) \|_{\sigma\tau} < 0) = \\
&\text{By (11)} \\
&[\| e_1 \|_{\sigma\tau}] + + \cdots + + [\| e_{i-1} \|_{\sigma\tau}] = \\
&\text{appling concatenation} \\
&[\| e_1 \|_{\sigma\tau}, \ldots, \| e_{i-1} \|_{\sigma\tau}] = \\
&\text{By Definition 2.4} \\
&\| [e_1, \ldots, e_{i-1}] \|_{\sigma\tau}
\end{aligned}
$$

3. $\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x > 0] \|_{\sigma\tau} = \| [e_j, \ldots, e_n] \|_{\sigma\tau}$ can be proven in similar way.

With this results we have that $\| a \|_{\sigma\tau} = \| [e_1, \ldots, e_n] \|_{\sigma\tau}$ and therefore:

$$
\| x \|_{x \mapsto e_j} = \| a[f] \|_{f \mapsto j} \tag{19}
$$

Now we can prove that for any set/list comprehension e with one generator, $\| \tau_s(\| e \|_\sigma) \|_\emptyset = \| \tau_s(e) \|_{\sigma\tau}$ and $\| \tau_e(\| e \|_\sigma) \|_\emptyset = \| \tau_e(e) \|_{\sigma\tau}$

* G_m is of the form g_m in ge_m with ge_m a set/array of standard type:

$$
\begin{aligned}
&\| \tau_s(\| e \|_\sigma) \|_\emptyset = \\
&\| \tau_s(\| \langle exp \mid G_m \text{ where } cond \rangle \|_\sigma) \|_\emptyset = \\
&\text{By Definition 2.6a} \\
&\|\tau_s(\mathcal{C}(\| exp \|_{\sigma \uplus g_m \mapsto e_1}, \| cond \|_{\sigma \uplus g_m \mapsto e_1}) \diamond \ldots \\
&\diamond \mathcal{C}(\| exp \|_{\sigma \uplus g_m \mapsto e_n}, \| cond \|_{\sigma \uplus g_m \mapsto e_n}))\|_\emptyset = \\
&\text{By function call transformation}
\end{aligned} \tag{20}
$$

$$\begin{aligned}
&\|\tau_s(\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_1}, \| cond \|_{\sigma\uplus g_m\mapsto e_1})) \diamond \ldots \\
&\diamond\tau_s(\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_n}, \| cond \|_{\sigma\uplus g_m\mapsto e_n}))\|_\emptyset = \\
&\text{Demonstrated bellow} \\
&\mathcal{C}(\| \tau_s(exp) \|_{\sigma\uplus g_m\mapsto e_1 \mathcal{T}}, \| \tau_g(cond) \|_{\sigma\uplus g_m\mapsto e_1 \mathcal{T}}) \diamond \ldots \\
&\diamond\mathcal{C}(\| \tau_s(exp) \|_{\sigma\uplus g_m\mapsto e_n \mathcal{T}}, \| \tau_g(cond) \|_{\sigma\uplus g_m\mapsto e_n \mathcal{T}}) = \\
&\text{By Definition 4 and identifier transformation,} \\
&\text{having } g_m \text{ is an identifier of standard type} \\
&\mathcal{C}(\| \tau_s(exp) \|_{\sigma^{\mathcal{T}}\uplus g_m\mapsto \tau_s(e_1)}, \| \tau_g(cond) \|_{\sigma^{\mathcal{T}}\uplus g_m\mapsto \tau_s(e_1)}) \diamond \ldots \\
&\diamond\mathcal{C}(\| \tau_s(exp) \|_{\sigma^{\mathcal{T}}\uplus g_m\mapsto \tau_s(e_n)}, \| \tau_g(cond) \|_{\sigma^{\mathcal{T}}\uplus g_m\mapsto \tau_s(e_n)}) = \\
&\text{By Definition 2.6a} \\
&\| \langle \tau_s(exp) \mid g_m \text{ in } \tau_s(ge_m) \text{ where } \tau_g(cond) \rangle \|_{\sigma^{\mathcal{T}}} = \\
&\| \tau_s(\langle exp \mid G_m \text{ where } cond \rangle) \|_{\sigma^{\mathcal{T}}} = \\
&\| \tau_s(e) \|_{\sigma^{\mathcal{T}}}
\end{aligned} \tag{21}$$

* G_m is of the form g_m in ge_m with ge_m a set/array of extended type:

$$\begin{aligned}
&\| \tau_s(\| e \|_\sigma) \|_\emptyset = \\
&\| \tau_s(\| \langle exp \mid G_m \text{ where } cond \rangle \|_\sigma) \|_\emptyset = \\
&\text{By Definition 2.6a}
\end{aligned}$$

$$\|\tau_s(\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_1}, \| cond \|_{\sigma\uplus g_m\mapsto e_1}) \diamond \ldots$$
$$\diamond\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_n}, \| cond \|_{\sigma\uplus g_m\mapsto e_n}))\|_\emptyset =$$
By function call transformation
$$\|\tau_s(\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_1}, \| cond \|_{\sigma\uplus g_m\mapsto e_1})) \diamond \ldots$$
$$\diamond\tau_s(\mathcal{C}(\| exp \|_{\sigma\uplus g_m\mapsto e_n}, \| cond \|_{\sigma\uplus g_m\mapsto e_n}))\|_\emptyset =$$
By (19)
$$\|\tau_s(\mathcal{C}(\| exp' \|_{\sigma\uplus f\mapsto 1}, \| cond' \|_{\sigma\uplus f\mapsto 1})) \diamond \ldots$$
$$\diamond\tau_s(\mathcal{C}(\| exp' \|_{\sigma\uplus f\mapsto n}, \| cond' \|_{\sigma\uplus f\mapsto n}))\|_\emptyset = \tag{22}$$
Demonstrated below
$$\mathcal{C}(\| \tau_s(exp') \|_{\sigma\uplus f\mapsto 1^\mathcal{T}}, \| \tau_g(cond') \|_{\sigma\uplus f\mapsto 1^\mathcal{T}}) \diamond \ldots$$
$$\diamond\mathcal{C}(\| \tau_s(exp') \|_{\sigma\uplus f\mapsto n^\mathcal{T}}, \| \tau_g(cond') \|_{\sigma\uplus f\mapsto n^\mathcal{T}}) =$$
By Definition 4 and constant and identifier transformation,
having f is an identifier of standard type
$$\mathcal{C}(\| \tau_s(exp') \|_{\sigma^\mathcal{T}\uplus f\mapsto 1}, \| \tau_g(cond') \|_{\sigma^\mathcal{T}\uplus f\mapsto 1}) \diamond \ldots$$
$$\diamond\mathcal{C}(\| \tau_s(exp') \|_{\sigma^\mathcal{T}\uplus f\mapsto n}, \| \tau_g(cond') \|_{\sigma^\mathcal{T}\uplus f\mapsto n}) =$$
By Definition 2.6a
$$\| \langle \tau_s(exp') \mid g_m \text{ in } G'_m \text{ where } \tau_g(cond')\rangle \|_{\sigma^\mathcal{T}} =$$
$$\| \tau_s(\langle exp \mid G_m \text{ where } cond\rangle) \|_{\sigma^\mathcal{T}} =$$
$$\| \tau_s(e) \|_{\sigma^\mathcal{T}}$$

Steps (21) and (22)can be performed because:

- For list comprehensions:
  - If $\| c \|_\sigma = false$:
    $\| c \|_\sigma = false$ iff $\| \tau_s(\| c \|_\sigma) \|_\emptyset = false$ iff $\| \tau_s(c) \|_{\sigma^\mathcal{T}} = false$

$$\| \tau_s(\mathcal{C}(\| e \|_\sigma, \| \tau_s(c) \|_\sigma)) \|_\emptyset = \text{By } \mathcal{C}(,) \text{ definition}$$
$$\| \tau_s([]) \|_\emptyset = \text{By array transformation}$$
$$\| [] \|_\emptyset = \text{By Definition 4}$$
$$[] = \text{By Definition 4}$$
$$\| [] \|_{\sigma^\mathcal{T}} = \text{By array transformation}$$
$$\| \tau_s(e) \|_{\sigma^\mathcal{T}} = \text{By } \mathcal{C}(,) \text{ definition}$$
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_s(c) \|_{\sigma^\mathcal{T}}) = \text{By } \tau_g(c) \text{ definition}$$
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_g(c) \|_{\sigma^\mathcal{T}})$$

· If $\| c \|_\sigma = true$:
$\| c \|_\sigma = true$ iff $\| \tau_s(\| c \|_\sigma) \|_\emptyset = true$ iff $\| \tau_s(c) \|_{\sigma^\mathcal{T}} = true$

$$
\begin{aligned}
\| \tau_s(\mathcal{C}(\| e \|_\sigma, \| \tau_s(c) \|_\sigma)) \|_\emptyset &= \text{By } \mathcal{C}(,) \text{ definition} \\
\| \tau_s([\| e \|_\sigma]) \|_\emptyset &= \text{By array transformation} \\
\| [\tau_s(\| e \|_\sigma)] \|_\emptyset &= \text{By Definition 2.4} \\
[\| \tau_s(\| e \|_\sigma) \|_\emptyset] &= \text{By I.H.} \\
[\| \tau_s(e) \|_{\sigma^\mathcal{T}}] &= \text{By } \mathcal{C}(,) \text{ definition} \\
\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_s(c) \|_{\sigma^\mathcal{T}}) &= \text{By } \tau_g(c) \text{ definition} \\
\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_g(c) \|_{\sigma^\mathcal{T}})
\end{aligned}
$$

* For set comprehensions:

· If $\| c \|_\sigma = false$:
$\| c \|_\sigma = false$ iff $\| \tau_s(\| c \|_\sigma) \|_\emptyset = false$ iff $\| \tau_s(c) \|_{\sigma^\mathcal{T}} = false$

$$
\begin{aligned}
\| \tau_s(\mathcal{C}(\| e \|_\sigma, \| \tau_s(c) \|_\sigma)) \|_\emptyset &= \\
\text{By } \mathcal{C}(,) \text{ definition} \\
\| \{\} \|_\emptyset &= \\
\text{By } \mathcal{C}(,) \text{ definition} \\
\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_s(c) \|_{\sigma^\mathcal{T}} \wedge \| \tau_e(e) \|_{\sigma^\mathcal{T}}) &= \\
\text{By Definition 2.9} \\
\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_s(c) \wedge \tau_e(e) \|_{\sigma^\mathcal{T}}) &= \\
\text{By } \tau_g(c) \text{ definition} \\
\mathcal{C}(\| \tau_s(e) \|_{\sigma^\mathcal{T}}, \| \tau_g(c) \|_{\sigma^\mathcal{T}})
\end{aligned}
$$

· If $\| c \|_\sigma = true$

$$\|\tau_s(\mathcal{C}(\| e \|_\sigma, \| c \|_\sigma))\|_\emptyset =$$
By $\mathcal{C}(,)$ definition
$$\| \tau_s(\{\| e \|_\sigma\}) \|_\emptyset =$$
By set transformation
$$\| \{[\tau_s(\| e \|_\sigma)][i] \mid i \; in \; 1..1 \text{ where } [\tau_e(\| e \|_\sigma)][i] = 0\} \|_\emptyset =$$
By (2)
$$\mathcal{C}(\| \tau_s(\| e \|_\sigma) \|_\emptyset, \| \tau_e(\| e \|_\sigma) \|_\emptyset = 0) =$$
Having $\| \tau_s(\| c \|_\sigma) \|_\emptyset = \text{true}$
$$\mathcal{C}(\| \tau_s(\| e \|_\sigma) \|_\emptyset, \| \tau_s(\| c \|_\sigma) \|_\emptyset \wedge \| \tau_e(\| e \|_\sigma) \|_\emptyset = 0) =$$
By I.H.
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\tau}, \| \tau_s(c) \|_{\sigma^\tau} \wedge \| \tau_e(e) \|_{\sigma^\tau} = 0) =$$
By Definition 2.2
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\tau}, \| \tau_s(c) \|_{\sigma^\tau} \wedge \| \tau_e(e) \|_{\sigma^\tau} = \| 0 \|_{\sigma^\tau}) =$$
By Definition 2.9
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\tau}, \| \tau_s(c) \|_{\sigma^\tau} \wedge \| \tau_e(e) = 0 \|_{\sigma^\tau}) =$$
By Definition 2.9
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\tau}, \| \tau_s(c) \wedge \tau_e(e) = 0 \|_{\sigma^\tau}) =$$
By $\tau_g(c)$ definition
$$\mathcal{C}(\| \tau_s(e) \|_{\sigma^\tau}, \| \tau_g(c) \|_{\sigma^\tau})$$

- If e has $m > 1$ generators let a' be $\langle exp \mid G_2, \ldots, G_m \text{ where } cond\rangle$ and $G_1 = g_1$ in ge_1 with $\| ge_1 \|_\sigma = \langle e_1, \ldots, e_n\rangle$:
  * If ge_1 is a set/array of standard type:

$$\| \tau_s(\| e \|_\sigma) \|_\emptyset =$$
$$\| \tau_s(\| \langle exp \mid G_1, \ldots, G_m \text{ where } cond\rangle \|_\sigma) \|_\emptyset = \text{By Definition 2.6b}$$
$$\|\tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond\rangle \|_{\sigma \uplus g_1 \mapsto e_1} \diamond \ldots \diamond$$
$$\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond\rangle \|_{\sigma \uplus g_1 \mapsto e_n})\|_\emptyset =$$
By predefined function transformation

$$\|\tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond\rangle \|_{\sigma \uplus g_1 \mapsto e_1}) \diamond \ldots \quad \text{(-12)}$$
$$\diamond \; \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond\rangle \|_{\sigma \uplus g_1 \mapsto e_n})\|_\emptyset = \quad \text{(-11)}$$
By Definition 2.9 (-10)

(-9)

$$\| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_1}) \|_\emptyset \diamond \ldots$$
$$\diamond \| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_n}) \|_\emptyset =$$
By I.H.
$$\| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma \uplus g_1 \mapsto e_1} \tau \diamond \ldots$$
$$\diamond \| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma \uplus g_1 \mapsto e_n} \tau =$$
By Definition 4 and identifier transformation,

with g_1 an identifier of standard type
$$\| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma^\tau \uplus g_1 \mapsto \tau_s(e_1)} \diamond \ldots$$
$$\diamond \| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma^\tau \uplus g_1 \mapsto \tau_s(e_n)} =$$
By definition2.6b
$$\| \langle \tau_s(exp') \mid g_1 \text{ in } \tau_s(ge_1), G'_2, \ldots, G'_m \text{ where } \tau_g(cond') \rangle \|_\sigma \tau =$$
By comprehension set/list transformation
$$\| \tau_s(\langle exp \mid G'_1, \ldots, G'_m \text{ where } cond \rangle) \|_\sigma \tau =$$
$$\| \tau_s(e) \|_\sigma \tau$$

* If ge_1 is a set/array of extended type:

$$\| \tau_s(\| e \|_\sigma) \|_\emptyset =$$
$$\| \tau_s(\| \langle exp \mid G_1, \ldots, G_m \text{ where } cond \rangle \|_\sigma) \|_\emptyset =$$
By Definition 2.6b
$$\| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_1} \diamond \ldots$$
$$\diamond \| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_n}) \|_\emptyset =$$
By predefined function transformation

$$\| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_1}) \diamond \ldots$$
$$\diamond\ \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_n}) \|_\emptyset = \quad \text{(-8)}$$
By Definition 2.9
$$\| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_1}) \|_\emptyset \diamond \ldots$$
$$\diamond \| \tau_s(\| \langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle \|_{\sigma \uplus g_1 \mapsto e_n}) \|_\emptyset =$$
By I.H.
$$\| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma \uplus g_1 \mapsto e_1} \tau \diamond \ldots$$
$$\diamond \| \tau_s(\langle exp \mid G_2, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma \uplus g_1 \mapsto e_n} \tau =$$

$$\begin{aligned}
&\text{By (2), having } g_1,\, g_n \text{ appear only in } exp \text{ and } cond\\
&\|\, \langle \tau_s(exp') \mid G'_2, \ldots, G'_m \text{ where } \tau_g(cond')\rangle \,\|_{\sigma \uplus f \mapsto 1}\tau \diamond \ldots\\
&\diamond \|\, \langle \tau_s(exp') \mid G'_2, \ldots, G'_m \text{ where } \tau_g(cond')\rangle \,\|_{\sigma \uplus f \mapsto n}\tau =\\
&\text{By definition2.6b}\\
&\|\, \langle \tau_s(exp') \mid f \text{ in } 1..n, G'_2 \ldots, G'_m \text{ where } \tau_s(cond')\rangle \,\|_{\sigma}\tau =\\
&\text{By comprehension set/list transformation}\\
&\|\, \tau_s(\langle exp \mid G_1, \ldots, G_m \text{ where } cond\rangle) \,\|_{\sigma}\tau =\\
&\|\, \tau_s(e) \,\|_{\sigma}\tau
\end{aligned}$$

$\|\, \tau_e(\|\, e \,\|_{\sigma}) \,\|_{\emptyset} = \|\, \tau_e(e) \,\|_{\sigma}\tau$ can be proved in similar way.

– e is of the form $if\ c\ then\ e_1\ else\ e_2\ endif$

- If $\|\, c \,\|_{\sigma} = true$:

$$\begin{aligned}
&\|\, c \,\|_{\sigma} = true \Leftrightarrow\\
&\|\, \tau_s(\|\, c \,\|_{\sigma}) \,\|_{\emptyset} = true \Leftrightarrow\\
&\|\, \tau_s(c) \,\|_{\sigma}\tau = true
\end{aligned} \tag{-7}$$

$$\begin{aligned}
&\|\, \tau_s(\|\, e \,\|_{\sigma}) \,\|_{\emptyset} =\\
&\|\, \tau_s(\|\, if\ c\ then\ e_1\ else\ e_2\ endif \,\|_{\sigma}) \,\|_{\emptyset} =\\
&\text{By Definition 2.??}\\
&\|\, \tau_s(\|\, e_1 \,\|_{\sigma}) \,\|_{\emptyset} =\\
&\text{By I.H.}\\
&\|\, \tau_s(e_1) \,\|_{\sigma}\tau =\\
&\text{By (-7) and Definition 2.??}\\
&\|\, if\ \tau_s(c)\ then\ \tau_s(e_1)\ else\ \tau_s(e_2)\ endif \,\|_{\sigma}\tau =\\
&\text{By conditional expressions transformation}\\
&\|\, \tau_s(if\ c\ then\ e_1\ else\ e_2\ endif) \,\|_{\sigma}\tau
\end{aligned}$$

- If $\|\, c \,\|_{\sigma} = false$

$$\begin{aligned}
&\|\, c \,\|_{\sigma} = false \Leftrightarrow\\
&\|\, \tau_s(\|\, c \,\|_{\sigma}) \,\|_{\emptyset} = false \Leftrightarrow\\
&\|\, \tau_s(c) \,\|_{\sigma}\tau = false
\end{aligned} \tag{-6}$$

$$
\begin{aligned}
& \| \tau_s(\| e \|_\sigma) \|_\emptyset = \\
& \| \tau_s(\| if\ c\ then\ e_1\ else\ e_2\ endif \|_\sigma) \|_\emptyset = \\
& \text{By Definition 2.??} \\
& \| \tau_s(\| e_2 \|_\sigma) \|_\emptyset = \\
& \text{By I.H.} \\
& \| \tau_s(e_2) \|_{\sigma^\mathcal{T}} = \\
& \text{By (-6) and Definition 2.??} \\
& \| if\ \tau_s(c)\ then\ \tau_s(e_1)\ else\ \tau_s(e_2)\ endif \|_{\sigma^\mathcal{T}} = \\
& \text{By conditional expressions transformation} \\
& \| \tau_s(if\ c\ then\ e_1\ else\ e_2\ endif) \|_{\sigma^\mathcal{T}}
\end{aligned}
$$

The prove for $\| \tau_e(\| e \|_\sigma) \|_\emptyset = \| \tau_e(e) \|_{\sigma^\mathcal{T}}$ is similar.

– e is a forall/exists expression of the form $e \equiv \mathit{forall/exists}(a)$
Let a be $[e_1, \ldots, e_n]$, then $\| a \|_\sigma = [\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma]$

$$
\begin{aligned}
\| \tau_s(\| e \|_\sigma) \|_\emptyset &= \\
\| \tau_s(\| forall(a) \|_\sigma) \|_\emptyset &= \text{By Definition 2.10} \\
\| \tau_s(\| e_1 \|_\sigma \wedge \cdots \wedge \| e_n \|_\sigma) \|_\emptyset &= \text{By function call transformation} \\
\| \tau_s(\| e_1 \|_\sigma) \wedge \cdots \wedge \tau_s(\| e_n \|_\sigma) \|_\emptyset &= \text{By Definition 2.9} \\
\| \tau_s(\| e_1 \|_\sigma) \|_\emptyset \wedge \cdots \wedge \| \tau_s(\| e_n \|_\sigma) \|_\emptyset &= \text{By I.H.} \\
\| \tau_s(e_1) \|_{\sigma^\mathcal{T}} \wedge \cdots \wedge \| \tau_s(e_n) \sigma^\mathcal{T} \|_\emptyset &= \\
\| forall([\tau_s(e_1), \ldots, \tau_s(e_n)]) \|_{\sigma^\mathcal{T}} &= \text{By Definition 2.10} \\
\| forall(\tau_s([e_1, \ldots, e_n])) \|_{\sigma^\mathcal{T}} &= \\
\| \tau_s(forall(a)) \|_{\sigma^\mathcal{T}} &=
\end{aligned}
$$

$$
\begin{aligned}
\| \tau_e(\| e \|_\sigma) \|_\emptyset &= \\
\| \tau_e(\| forall(a) \|_\sigma) \|_\emptyset &= \\
\| \tau_e(\| e_1 \|_\sigma \wedge \cdots \wedge \| e_n \|_\sigma) \|_\emptyset &= \\
\| 0 \|_\emptyset & \\
\| 0 \|_{\sigma^\mathcal{T}} &= \\
\| \tau_e(forall(a)) \|_{\sigma^\mathcal{T}} &
\end{aligned} \tag{-5}
$$

Notice (-5) is done because $\Gamma \vdash a ::< array\ of\ bool >$ and therefore $\| e_1 \|_\sigma \wedge \cdots \wedge \| e_n \|_\sigma$ is a Boolean expression and its τ_e transformation is 0.

Finally, we can establish the theoretical result.

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

Proof Idea

The result is a consequence of a similar result applied to expressions:
For every expression e and well-typed substitution σ:

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

where id represents the identity substitution. These results can be proven using

Proof. According to Definition 3, we must prove:
1.- For every assignment a of the form $id = e$ in $\mathcal{M}$, $\| a \|_\sigma = true$ iff for each assignment a' in $\mathcal{M}^\mathcal{T}$, $\| a' \|_{\sigma^\mathcal{T}} = true$ (Definition 1):

1. Assume all the assignments a of the form $id = e$ in $\mathcal{M}$, verify $\| a \|_\sigma = true$. Let a' be an assignment in $\mathcal{M}^\mathcal{T}$. We prove $\| a' \|_{\sigma^\mathcal{T}} = true$:
   By the definition of the transformation a' in $\mathcal{M}^\mathcal{T}$ comes from the transformation of an assignment $id = e$ in $\mathcal{M}$. We distinguish cases depending on the type of id:

   – If $\Gamma \ \vdash id :: t \wedge st(t)$ then by transformation definition a' is of the form $id = \tau_s(e)$.

   $$\begin{aligned} \| a \|_\sigma = true &\Rightarrow \\ \| id = e \|_\sigma = true &\Rightarrow \text{By Definition 2.8} \\ \| id \|_\sigma = \| e \|_\sigma &\Rightarrow \text{Applying } \| \tau_s() \|_\emptyset \text{ both sizes} \\ \| \tau_s(\| id \|_\sigma) \|_\emptyset = \| \tau_s(\| e \|_\sigma) \|_\emptyset &\Rightarrow \text{By Lemma 3} \\ \| \tau_s(id) \|_{\sigma^\mathcal{T}} = \| \tau_s(e) \|_{\sigma^\mathcal{T}} &\Rightarrow \text{By transformation} \\ \| id \|_{\sigma^\mathcal{T}} = \| \tau_s(e) \|_{\sigma^\mathcal{T}} &\Rightarrow \text{By Definition 2.8} \\ \| id = \tau_s(e) \|_{\sigma^\mathcal{T}} = true & \end{aligned}$$

   – If $\Gamma \ \vdash id :: t \wedge et(t)$ then by transformation definition a' is of the form $\tau_t(id) = \tau_t(e)$ with t being either s or e.

   $$\begin{aligned} \| a \|_\sigma = true &\Rightarrow \\ \| id = e \|_\sigma = true &\Rightarrow \text{By Definition 2.8} \\ \| id \|_\sigma = \| e \|_\sigma &\Rightarrow \text{Applying } \| \tau_t() \|_\emptyset \text{ both sizes} \\ \| \tau_t(\| id \|_\sigma) \|_\emptyset = \| \tau_t(\| e \|_\sigma) \|_\emptyset &\Rightarrow \text{By Lemma 3} \\ \| \tau_t(id) \|_{\sigma^\mathcal{T}} = \| \tau_t(e) \|_{\sigma^\mathcal{T}} &\Rightarrow \text{By Definition 2.8} \\ \| \tau_t(id) = \tau_t(e) \|_{\sigma^\mathcal{T}} = true & \end{aligned}$$

2. Assume all the assignments a' of the form $id' = e'$ in $\mathcal{M}^\mathcal{T}$, verify $\| a' \|_\sigma = true$. Let a be an assignment in $\mathcal{M}$. We prove $\| a \|_\sigma = true$:
   By the definition of the transformation a' in $\mathcal{M}^\mathcal{T}$ comes from the transformation of an assignment $id = e$ in $\mathcal{M}$. We distinguish cases depending on the type of id:

– If $\Gamma \vdash id :: t \wedge st(t)$ then by transformation definition a' is of the form $id = \tau_s(e)$.

$$\begin{aligned}
\| a' \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \\
\| id = \tau_s(e) \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \text{By Definition 2.8} \\
\| id \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By identifier transformation} \\
\| \tau_s(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2. 1} \\
\tau_s(id)\sigma^{\mathcal{T}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Lemma 3} \\
\tau_s(id)\sigma^{\mathcal{T}} = \| \tau_s(\| e \|_{\sigma}) \|_{\emptyset} &\Rightarrow \\
\tau_s(id)\sigma^{\mathcal{T}} = \tau_s(\| e \|_{\sigma}) &\Rightarrow \\
\tau_s(id) \mapsto \tau_s(\| e \|_{\sigma}) \in \sigma^{\mathcal{T}} &\Rightarrow \text{By Definition 4} \\
id \mapsto \| e \|_{\sigma} \in \sigma &\Rightarrow \\
\| id \|_{\sigma} = \| e \|_{\sigma} &\Rightarrow \text{By Definition 2.8} \\
\| id = e \|_{\sigma} = true &
\end{aligned}$$

– If $\Gamma \vdash id :: t \wedge et(t)$ then by transformation definition a' is of the form $\tau_t(id) = \tau_t(e)$, with t either s or e.

$$\begin{aligned}
\| a' \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \\
\| \tau_t(id) = \tau_t(e) \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \text{By Definition 2.8} \\
\| \tau_t(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_t(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2.1} \\
\tau_t(id)\sigma^{\mathcal{T}} = \| \tau_t(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Lemma 3} \\
\tau_t(id)\sigma^{\mathcal{T}} = \| \tau_t(\| e \|_{\sigma}) \|_{\emptyset} &\Rightarrow \\
\tau_t(id)\sigma^{\mathcal{T}} = \tau_t(\| e \|_{\sigma}) &\Rightarrow \\
\tau_t(id) \mapsto \tau_t(\| e \|_{\sigma}) \in \sigma^{\mathcal{T}} &\Rightarrow \text{By Definition 4} \\
id \mapsto \| e \|_{\sigma} \in \sigma &\Rightarrow \\
\| id \|_{\sigma} = \| e \|_{\sigma} &\Rightarrow \text{By Definition 2.8} \\
\| id = e \|_{\sigma} = true &
\end{aligned}$$

2.- Every *constraint* c in $\mathcal{M}$ verifies $\| c \|_{\sigma} = true$ iff every *constraint* c' in $\mathcal{M}^{\mathcal{T}}$, $\| c' \|_{\sigma^{\mathcal{T}}} = true$:

– constraint c $\in \mathcal{M}$ iff constraint $\tau_s(c) \in \mathcal{M}^{\mathcal{T}}$
  By transformation definition, constraint c $\in \mathcal{M}$ iff constraint $\tau_s(c) \in \mathcal{M}^{\mathcal{T}}$.
  $\| c \|_{\sigma} = true$ iff $\| \tau_s(\| c \|_{\sigma}) \|_{\emptyset} = true$ iff, By Lemma 3, $\| \tau_s(c) \|_{\sigma^{\mathcal{T}}} = true$

8 Prototype

We illustrate the usage of MiniZinc+ using automatically generated[7] models for generating SQL test cases for views, taking into account NULL values. The models make use of extended integers and Booleans.

[7] Models generated by STCG, the tool presented in [**?**]

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

We show the results of the following two examples:

1. Model *Sql Or* represents a simple example of SQL test cases. The model represents the possible data in an SQL database with only one row in its table T such that the view V is not empty for the following SQL code:

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

   with the following MiniZinc+ code:

```
var intE: T_c_0;
var intE: T_b_0;
var intE: T_a_0;
constraint (true /\ ((T_a_0 != T_b_0) \/ (T_a_0 != T_c_0)));
solve satisfy;
output [" INSERT INTO T (a,b,c) VALUES (", show(T_a_0),",", show
    (T_b_0),",", show(T_c_0),");\n"];
```

2. Model *Board* represents the possible data in a more involved SQL database example presented in [?]:

```
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)));
```

   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. It is modeled by the following minizinc code:

```
var intE: player_id_1;
var intE: player_id_0;
var intE: board_y_1;
var intE: board_y_0;
var intE: board_x_1;
var intE: board_x_0;
var intE: board_id_1;
var intE: board_id_0;
% Table constraints for table player:
constraint ((true /\ (true /\ (player_id_0 != player_id_1))) /\ true)
    ;
constraint ((((true /\ (true /\ (player_id_0 != player_id_1))) /\
    true) /\ ((((((true /\ (true /\ (player_id_0 != player_id_1)))
    /\ true) /\ (((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (
    board_y_0 != board_y_1)))) /\ true) /\ (((board_id_0 =
    player_id_0) \/ (board_id_0 = player_id_1)) /\ ((board_id_1 =
    player_id_0) \/ (board_id_1 = player_id_1)))) /\ (board_id_0 =
    player_id_0)) \/ ((((true /\ (true /\ ((board_x_0 != board_x_1)
    \/ (board_y_0 != board_y_1)))) /\ true) /\ (((board_id_0 =
    player_id_0) \/ (board_id_0 = player_id_1)) /\ ((board_id_1 =
    player_id_0) \/ (board_id_1 = player_id_1)))) /\ (board_id_1 =
    player_id_0)))) /\ (player_id_0 = player_id_0)) \/ ((((true /\ (
    true /\ (player_id_0 != player_id_1))) /\ true) /\ (((((true /\
    (true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))
    ) /\ true) /\ (((board_id_0 = player_id_0) \/ (board_id_0 =
    player_id_1)) /\ ((board_id_1 = player_id_0) \/ (board_id_1 =
    player_id_1)))) /\ (board_id_0 = player_id_1)) \/ ((((true /\ (
    true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1))))
     /\ true) /\ (((board_id_0 = player_id_0) \/ (board_id_0 =
    player_id_1)) /\ ((board_id_1 = player_id_0) \/ (board_id_1 =
    player_id_1)))) /\ (board_id_1 = player_id_1)))) /\ (player_id_1
     = player_id_0)))
 /\ (not (((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (
     board_y_0 != board_y_1)))) /\ true) /\ (((board_id_0 =
     player_id_0) \/ (board_id_0 = player_id_1)) /\ ((board_id_1 =
     player_id_0) \/ (board_id_1 = player_id_1)))) /\ ((board_id_0 =
      player_id_0) /\ (not (((((true /\ (true /\ ((board_x_0 !=
     board_x_1) \/ (board_y_0 != board_y_1)))) /\ true) /\ (((
     board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\ ((
     board_id_1 = player_id_0) \/ (board_id_1 = player_id_1)))) /\
     (((board_x_0 = board_x_0) \/ (board_y_0 = board_y_0)) /\ (
     board_id_0 != board_id_0))) \/ ((((true /\ (true /\ ((board_x_0
      != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true) /\ (((
     board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\ ((
     board_id_1 = player_id_0) \/ (board_id_1 = player_id_1)))) /\
     (((board_x_1 = board_x_0) \/ (board_y_1 = board_y_0)) /\ (
     board_id_0 != board_id_1))))))) \/ ((((true /\ (true /\ ((
     board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)
      /\ (((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1))
      /\ ((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1)))
     ) /\ ((board_id_1 = player_id_0) /\ (not (((((true /\ (true /\
     ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\
     true) /\ (((board_id_0 = player_id_0) \/ (board_id_0 =
     player_id_1)) /\ ((board_id_1 = player_id_0) \/ (board_id_1 =
     player_id_1)))) /\ (((board_x_0 = board_x_1) \/ (board_y_0 =
     board_y_1)) /\ (board_id_1 != board_id_0))) \/ ((((true /\ (
     true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))
     ) /\ true) /\ (((board_id_0 = player_id_0) \/ (board_id_0 =
     player_id_1)) /\ ((board_id_1 = player_id_0) \/ (board_id_1 =
     player_id_1)))) /\ (((board_x_1 = board_x_1) \/ (board_y_1 =
     board_y_1)) /\ (board_id_1 != board_id_1)))))))))) \/
(((true /\ (true /\ (player_id_0 != player_id_1))) /\ true) /\ ((((((
    true /\ (true /\ (player_id_0 != player_id_1))) /\ true) /\
    (((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (board_y_0 !=
     board_y_1)))) /\ true) /\ (((board_id_0 = player_id_0) \/ (
    board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/ (
    board_id_1 = player_id_1)))) /\ (board_id_0 = player_id_0)) \/
    ((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (board_y_0 !=
```

```
     board_y_1)))) /\ true) /\ (((board_id_0 = player_id_0) \/ (
     board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/ (
     board_id_1 = player_id_1)))) /\ (board_id_1 = player_id_0)))) /\
      (player_id_0 = player_id_1)) \/ ((((true /\ (true /\ (
     player_id_0 != player_id_1))) /\ true) /\ (((((true /\ (true /\
     ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true
     ) /\ (((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1))
      /\ ((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))))
      /\ (board_id_0 = player_id_1)) \/ ((((true /\ (true /\ ((
     board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)
     /\ (((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1))
     /\ ((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))))
     /\ (board_id_1 = player_id_1)))) /\ (player_id_1 = player_id_1))
     ) /\
 (not (((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (board_y_0 !=
      board_y_1)))) /\ true) /\ (((board_id_0 = player_id_0) \/ (
     board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/ (
     board_id_1 = player_id_1)))) /\ ((board_id_0 = player_id_1) /\ (
     not (((((true /\ (true /\ ((board_x_0 != board_x_1) \/ (
     board_y_0 != board_y_1)))) /\ true) /\ (((board_id_0 =
     player_id_0) \/ (board_id_0 = player_id_1)) /\ ((board_id_1 =
     player_id_0) \/ (board_id_1 = player_id_1)))) /\ (((board_x_0 =
     board_x_0) \/ (board_y_0 = board_y_0)) /\ (board_id_0 !=
     board_id_0))) \/ ((((true /\ (true /\ ((board_x_0 != board_x_1)
     \/ (board_y_0 != board_y_1)))) /\ true) /\ (((board_id_0 =
     player_id_0) \/ (board_id_0 = player_id_1)) /\ ((board_id_1 =
     player_id_0) \/ (board_id_1 = player_id_1)))) /\ (((board_x_1 =
     board_x_0) \/ (board_y_1 = board_y_0)) /\ (board_id_0 !=
     board_id_1))))))) \/ ((((true /\ (true /\ ((board_x_0 !=
     board_x_1) \/ (board_y_0 != board_y_1)))) /\ true) /\ (((
     board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\ ((
     board_id_1 = player_id_0) \/ (board_id_1 = player_id_1)))) /\ ((
     board_id_1 = player_id_1) /\ (not (((((true /\ (true /\ ((
     board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)
     /\ (((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1))
     /\ ((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))))
     /\ (((board_x_0 = board_x_1) \/ (board_y_0 = board_y_1)) /\ (
     board_id_1 != board_id_0))) \/ ((((true /\ (true /\ ((board_x_0
     != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true) /\ (((
     board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\ ((
     board_id_1 = player_id_0) \/ (board_id_1 = player_id_1)))) /\
     (((board_x_1 = board_x_1) \/ (board_y_1 = board_y_1)) /\ (
     board_id_1 != board_id_1)))))))))));
 solve satisfy;
 output ["INSERT INTO board (id,x,y) VALUES (", show(board_id_0),",",
      show(board_x_0),",", show(board_y_0),");\n","INSERT INTO board
      (id,x,y) VALUES (", show(board_id_1),",", show(board_x_1),",",
     show(board_y_1),");\n","INSERT INTO player (id) VALUES (", show
     (player_id_0),");\n","INSERT INTO player (id) VALUES (", show(
     player_id_1),");\n"];
```

Both models include the definitions of the functions (=,!=,∨, ∧) 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))),
    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;
```

The size, translation time and solving times of this models is shown in the following table, compared with the same problem without considering NULL values modelled in standard MiniZinc.

model	variables	function calls	size (KB)	translation time (ms)	solve time (ms)
Sql Or	5	4	0.483	17	0.498
Sql Or$^+$	99	254	5.009	-	0.291
Board	54	419	13.269	2923	0.326
Board$^+$	2032	374678	15946.193	-	2.209

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

9 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.[8]

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.

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.

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

[8] Although for these scheduling problems there are approaches [7] which support stronger propagation.

4. 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.
5. IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, Aug. 1985.
6. L. D. Koninck, S. Brand, and P. J. Stuckey. Constraints in non-boolean contexts. In *ICLP (Technical Communications)*, pages 117–127, 2011.
7. 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.
8. G. Malinowski. *Many-Valued Logics*. Oxford University Press, 1993.
9. 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.
10. 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.