MiniZinc^{+}: Finite Type Extensions for Constraint Programming Language MiniZinc

Antonio Tenorio Fornés

Máster en Investigación en Informática

Facultad de Informática

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Trabajo Fin de Máster en Programación y Tecnología Software

Curso: 2012-2013

Director: Rafael Caballero Roldán
Colaborador Externo: Peter J. Stuckey

Convocatoria: Septiembre 2013
Calificación: 7

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: " MiniZinc^{+}*: Finite Type Extensions for Constraint Programming Language* MiniZinc" , realizado durante el curso académico 2012-2013 bajo la dirección de Rafael Caballero Roldán y con la colaboración externa de dirección de Peter J. Stuckey en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Antonio Tenorio Fornés

Madrid, 27 de septiembre de 2013

Resumen

Muchos problemas se modelan de forma natural extendiendo un tipo existente con valores adicionales. Por ejemplo, los tipos de las bases de datos se extienden con valores nulos para representar la falta de datos. De forma parecida, los enteros pueden extenderse para tratar $-\infty$ y $+\infty$. Estas extensiones no solo afectan a la representación de tipos sino también a las funciones y operaciones entre los elementos del nuevo tipo extendido. Por ejemplo, para extender el tipo entero con los valores $-\infty$ y $+\infty$, las operaciones aritméticas como la suma y la resta se deben redefinir para los nuevos valores. Nuestra propuesta extiende el lenguaje de programación con restricciones MiniZinc para permitir modelos con tipos extendidos. El sistema permite la definición de tipos extendidos y del comportamiento de las operaciones respecto a los nuevos tipos. Los modelos extendidos son transformados a modelos MiniZinc equivalentes. Definimos la semántica tanto de MiniZinc como de la extensión propuesta y probamos la corrección de la transformación. El uso y las aplicaciones del nuevo lenguaje se ilustran a través de ejemplos incluyendo problemas SQL, circuitos Booleanos con posibles entradas indefinidas y problemas de planificación con valores especiales para el tiempo.

Palabras Clave

Extensión de tipos, Programación con Restricciones, Transformación de Programas, Lenguajes Declarativos, MiniZinc, Satisfacción de Restricciones, Optimización.

Abstract

Many problems are naturally modeled by extending an existing type with additional values. For example, database types are extended with null values to represent missing data. Similarly, integers may be extended to handle $-\infty$ and $+\infty$. These extensions does not only affects the type representation but also the functions and operations involving elements of the new extended type. For instance, in order to extend integer type with $-\infty$ and $+\infty$ values, the arithmetical operations such as addition and subtraction must be defined for the new values. Our proposal extends the constraint modeling language MiniZinc to allow models with extended types. The framework allows the definition of the extended types and the behavior of the operations with relation to the new types. The extended models are transformed into equivalent MiniZinc models. We define the semantics of both MiniZinc and the proposed extension and prove the correctness of the transformation. The usage and applications of the new language are illustrated through examples including SQL like problems, Boolean circuits allowing undefined inputs and scheduling problems considering special time values.

Keywords

Type extension, Constraint programming, Program transformation, Declarative Languages, MiniZinc, Constraint Satisfaction, Optimization.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Approach 2
1.3 Applications 2
1.4 Contributions 3
1.5 Structure of the work 4

2 From MiniZinc to MiniZinc$^+$ **5**
2.1 MiniZinc with functions 5
2.1.1 Grammar 5
2.1.2 Types 7
2.1.3 Semantics 9
2.1.4 Examples 12
2.2 MiniZinc$^+$ 13
2.2.1 Syntactic Extension 14
2.2.2 Semantic Extension 14
2.2.3 Examples in MiniZinc$^+$ 14

3 From MiniZinc$^+$ to MiniZinc **21**
3.1 Transformation 21
3.1.1 Notation 22
3.1.2 Identifiers, constants, array and set expressions 22
3.1.3 Array and set comprehensions 24
3.1.4 Conditional and logical expressions 25
3.1.5 Predefined function and predicate calls 25
3.1.6 Declarations of extended types 26
3.1.7 Declarations of variables and parameters 26
3.1.8 Assignments and Constraints 27
3.1.9 Output Item 27

3.1.10 Optimization 28
3.2 Correctness 29

4 Prototype **47**
4.1 Functionalities 47
4.2 Architecture and Implementation 48
4.3 Manual 48
4.4 Conclusions and future development 49

5 Practical Application: SQL Test Case Generation **51**
5.1 SQL Test Case Generator 51
5.2 Extending the model with NULL values. 52
5.3 Experimental Results 54

6 Conclusions and future work **57**
6.1 Conclusions 57
6.2 Future Work 58

A MiniZinc$^+$ code of Model "Board" **61**

B Model "SQL_or" transformation **65**

List of Figures

2.1 A n bit full adder circuit 12
2.2 A n bit adder in MiniZinc: $x + y = s$ (variable declarations) . . 16
2.3 A n bit adder in MiniZinc: $x + y = s$ (constraints) 17
2.4 Truth tables including the undefined value 18
2.5 Modeling time with an extended value.. 18
2.6 A n bit adder in MiniZinc$^+$: $x + y = s$ 19
2.7 Modelling time with an extended value.. 20

Chapter 1

Introduction

This chapter starts introducing the motivation (Section 1.1) and the adopted approach (Section 1.2) of our project. Then, it shows a set of applications (Section 1.3) of the proposal and the main contributions of the project (Section 1.4). Finally, we describe the structure of the work (Section 1.5).

1.1 Motivation

Constraint Programming languages as MiniZinc [1] allows to express constraint satisfaction problems in terms of the involved variables and the restrictions they must satisfy. MiniZinc provides a helpful abstraction level to model constraint satisfaction problems for the standard types it was designed for (integers, Boolean, etc.).

This work propose an extension of the Constraint Programming language MiniZinc to allow the extension of the original types with a finite number of additional values. The idea was originally motivated by the need of extending MiniZinc types with NULL to represent SQL problems, not a trivial task as shown in Chapter 5.

As databases extend their types with additional values, there are other areas where the type extension is used as a data modeling resource: three valued and many-valued logic or floating point arithmetic are some of the examples. This domains share the modeling difficulties found for SQL: the modeler have to explicitly deal with the additional values behavior in each subexpression (See example in Section 2.1.4).

Our proposal provides an abstraction level to deal with type extensions, aiming to improve the quality of the models in terms of development time, error rate and readability.

1.2 Approach

The project presents MiniZinc$^+$, an extension of the Constraint Programming language MiniZinc which allows the extension of its types with a finite number of constants.

The adopted approach for extending the language can be divided in the following steps:

- First, the *syntax* of MiniZinc$^+$ is defined as an extension of MiniZinc with functions [2] syntax. This new syntax allow new type declarations.
- Then we define a formal semantic for MiniZinc. This semantic is adopted by MiniZinc$^+$, considering that the variables and parameters of an extended type can take extended constants as values.
- A transformation from MiniZinc$^+$ to MiniZinc is proposed. The main idea behind the transformation is to map each extended variable to two variables, representing either the standard or extended values the variable can take. The transformation also deals with more complex data structures as arrays or sets and more complex expressions as list and set comprehensions.
- The transformation is proven to be correct with respect to the proposed semantic. The model expressed in MiniZinc$^+$ is proven to be semantically equivalent to its MiniZinc transformation.

1.3 Applications

The extension of already existing types with a finite set of constants and the consequent function redefinitions is used in a wide set of disciplines:

- Arithmetic extended with positive and negative infinity:

  Of course infinity is not a number but a mathematical concept, but modern languages such as Java include the constant NaN representing an undefined or unrepresentable value, especially in floating-point calculations [3]. The treatment of NaN has several interesting properties: for instance is the only value of a numeric type such that is not the same as itself when compared. In fact the usual Java test to check if a number x is NaN is the following: boolean isNaN(x)return x != x;

  Our framework allows including this constant in constraint programming. The use of this concept applied to undefinedness in Constraint Programing and it's different semantics can yield interesting results [2].

- Boolean Logic:

  The many valued logic[4] was first introduced by Charles S. Peirce and later developed by Jean Lukasiewicz [5]. There are studies about the

definition of many valued logic, their applications to Logic Programing and to Constraint Logic Programming relevant for our project:

  - Azevedo propose the use of many valued logics in constraint programming for testing and diagnosis of digital systems [6]. Our work can be directly applied to his proposal, as we show in Section 2.2.3.
  - "MUltilog" [7] is a system that takes the definition of multi-valued logics and generates their sequent calculus, a natural deduction system, and clause formation rules. This system allows, as our proposal, the definition of extended Boolean types, but does not handle other type extensions or complex structures and it can not be used as a programming language for the extended domain.
  - Faye F. Liu and Douglas H. Moore [8] presents a system that uses a three valued logic for logic programming, explicitly handling indetermination and contradiction with additional values. This system is implemented for Logic Programing, but not for Constraint Logic Programing.

- Databases: Databases use one or more additional values for their types to model undefined or unaplicable data [9]. Modeling this kind of problems was the original motivation for the project.

To model problems with this extended types in Constraint Programming languages as MiniZinc is not trivial and therefore is a time consuming and error-prone task. The correct (See section 3.2), fully automatic method we propose is a valuable tool to model problems with extended types.

1.4 Contributions

The main contributions of the project are:

- The definition of MiniZinc$^+$, a new Constraint Programming language derived from MiniZinc that allows the extension of MiniZinc types with a finite set of constants.
- The description of a transformation from MiniZinc$^+$ to MiniZinc.
- The proof of the correctness of the proposed transformation.
- The implementation of the proposal in a working prototype.

The development of this contributions have also produce the following auxiliary results:

- The formalization of MiniZinc and MiniZinc$^+$ semantics.
- The formalization of MiniZinc and MiniZinc$^+$ the type inference rules.

1.5 Structure of the work

This work is structured in six chapters. This chapter introduces the proposal, showing its interest, applications and contributions as well as the adopted approach. Next chapter presents MiniZinc language and the characteristics of the proposed extension MiniZinc$^+$. Chapter 3 proposes a program transformation from our extended language to the reference language, demonstrates its correctness and illustrate the process with some examples. The theoretical ideas of the work have been implemented in the prototype presented in Chapter 4. Non-trivial real examples of the tool's applications and experimental results performed to measure the performance of the prototype and the feasibility of the proposal can be found in Chapter 5. Finally, Chapter 6 concludes this document, summarizing the contributions of this work and identifying future work.

Chapter 2

From MiniZinc to MiniZinc$^+$

This chapter guides the reader through the construction of MiniZinc$^+$, describing the source language MiniZinc and introducing the proposed extensions which define the new language.

First, MiniZinc with functions, the language we extend and the target language of the transformation we propose, is described. The syntax (Section 2.1.1), type inference rules (Section 2.1.2) and semantics (Section 2.1.3) of the language are provided. Some examples complete this presentation (Section 2.1.4).

Next, MiniZinc$^+$ is introduced by the description of the proposed extensions (Section 2.2). The MiniZinc examples from Section 2.1.4 are rewritten in the new language in Section 2.2.3, allowing the comparison among them.

2.1 MiniZinc with functions

MiniZinc [1] is a medium-level constraint modeling language that allows the modeler to express constraint problems easily. In particular we take as starting point the version of MiniZinc with functions described in [2] which is an extension that allows the use of user defined functions. Following sections define the language's syntax, types and semantic, illustrating the language presentation with examples.

2.1.1 Grammar

This section introduces MiniZinc with functions [2] grammar. A MiniZinc model consist of:

- Variable declarations (nonterminal decl).
- Assignments (nonterminal assig).
- Predicate and function definitions (nonterminals pred and funct)

- Constraints (nonterminal const).
- Solve statement (either solve satisfy for Constraint Satisfaction Problems or solve maximize/minimize exp for Optimization Problems)
- Output format statement.

The variables and expressions in MiniZinc can have int, bool or float types or be an array or set of this types. Array and set expressions (nonterminals arrexp and setexp) can be also expressed as array or set comprehension.

The following table formalizes MiniZinc with functions syntax:

```
exp     ⟶  vId | constant | vId[exp]
              | arrexp[exp] | setexp | arrexp
              | if exp then exp else exp endif
              | pId(exp*[,]) | fId(exp*[,])
              | let {decl*[,] const*[,]} in exp
              | forall (arrexp)
              | exists (arrexp)
arrexp  ⟶  [exp*[,]]
              | [exp | genvar+[,] where exp]
setexp  ⟶  { exp*[,] } | range
              | {exp | genvar+[,] where exp}
genvar  ⟶  vId+[,] in setexp | vId+[,] in arrexp
range   ⟶  exp .. exp
decl    ⟶  vtype : vId
              | array[range] of vtype : vId
              | set of type: vId
              | var set of setexp: vId
assig   ⟶  vId = exp
const   ⟶  constraint exp
funct   ⟶  function decl (decl*[,]) = exp
pred    ⟶  predicate pId(decl*[,]) = exp
solv    ⟶  solve satisfy | solve minimize vId
              | solve maximize vId
out     ⟶  output ([ sh*[,] ])
sh      ⟶  show(exp) | "string"
type    ⟶  int | bool | float | range
vtype   ⟶  type | var type
model   ⟶  decl,*[;]; assig*[;]; pred*[;]
              ; funct*[;];const*[;]; solv; out;
```

where model is the start symbol of the grammar, vId, fId and pId are identifiers for: parameters and variables, functions and predicates, respectively and **string** represents an arbitrary string constant. The values c_i represent new constant identifiers. The notation $n^{*[s]}$ / $n^{+[s]}$ indicates zero or more / one or more repetitions of the nonterminal “n” such that these repetitions are separated by string s. Boldface words are reserved words of the language.

A correct MiniZinc model should also be well-typed following the type inference rules introduced in next section.

2.1.2 Types

In this section we propose the type inference rules for MiniZinc models. This rules are used to check the type correctness of the model and as a resource for the formalization of MiniZinc semantics in next section. The context Γ is considered to have the type information of variables, constants, predicates and functions, that can be directly obtained from their declarations.

Following rules formalize the type inference rules of a MiniZinc Model.

Trivial :

$$\frac{e :: t \in \Gamma}{\Gamma \vdash e :: t}$$

Subtypes An order is defined among types, this order is used to check function calls type correctness. This order establish that:

- Any type $\langle t \rangle$ is "smaller or equal" than the type $\langle var\ t \rangle$:

$$\frac{}{\langle t \rangle <: \langle var\ t \rangle}$$

- Any type $\langle t \rangle$ is "smaller or equal" than itself:

$$\frac{}{t <: t}$$

Assignments An assignment expression is well typed if it is among expressions typed with the same type or if the type of one of the expressions is $\langle t \rangle$ and the other is $\langle var\ t \rangle$:

if $(t_1 <: t_2 \vee t_2 <: t_1)$: $$\frac{\Gamma \vdash e_1 :: t_1 \quad \Gamma \vdash e_2 :: t_2}{\vdash e_1 = e_2}$$

Comprehension generators The generators in set and array comprehensions (nonterminal genvar) have the following type rule:

$$\frac{\Gamma \vdash e :: \langle \text{set of } \mathsf{type} \rangle \vee \Gamma \vdash e :: \langle \text{array of } \mathsf{type} \rangle}{\Gamma, v_1 :: \langle \mathsf{type} \rangle, \ldots, v_n :: \langle \mathsf{type} \rangle \ \vdash v_1, \ldots, v_n\ in\ e}$$

Arrays :

- Arrays of the form $[e_1, \ldots, e_n]$:

$$\frac{\Gamma \vdash e_1 :: \langle \mathsf{vtype} \rangle \quad \ldots \quad \Gamma \vdash e_n :: \langle \mathsf{vtype} \rangle}{\Gamma \vdash [e_1, \ldots, e_n] :: \langle \text{array of } \mathsf{vtype} \rangle}$$

- Array comprehension:

$$\frac{\Gamma \vdash e_1 :: \langle \mathsf{vtype} \rangle \quad \Gamma \vdash \mathsf{genvars} \quad \Gamma \vdash \mathsf{cond} :: \langle bool \rangle}{\Gamma \vdash [e_1 \mid \mathsf{genvars}\ where\ \mathsf{cond}] :: \langle \text{array of } \mathsf{vtype} \rangle}$$

Array access :

$$\frac{\Gamma \vdash a :: \langle \text{array of } t \rangle \quad \Gamma \vdash e :: \langle int \rangle}{\Gamma \vdash a[e] :: \langle t \rangle}$$

Sets :

- Sets of the form $\{e_1, \ldots, e_n\}$:
$$\frac{\Gamma \vdash e_1 :: \langle \mathsf{type} \rangle \quad \ldots \quad \Gamma \vdash e_n :: \langle \mathsf{type} \rangle}{\Gamma \vdash \{e_1, \ldots, e_n\} :: \langle \text{set of } \mathsf{type} \rangle}$$
- Set comprehension:
$$\frac{\Gamma \vdash e_1 :: \langle \mathsf{type} \rangle \quad \Gamma \vdash \mathsf{genvars} \quad \Gamma \vdash \mathsf{cond} :: \langle bool \rangle}{\Gamma \vdash \{e_1 \mid \mathsf{genvars}\ \textit{where}\ \mathsf{cond}\} :: \langle \text{set of } \mathsf{type} \rangle}$$
- Ranges:
$$\frac{\Gamma \vdash e_1 :: \langle int \rangle \quad \Gamma \vdash e_2 :: \langle int \rangle}{\Gamma \vdash e_1..e_2 :: \langle \text{set of int} \rangle}$$

Exists and Forall :
$$\frac{\Gamma \vdash e :: \langle \text{array of bool} \rangle}{\Gamma \vdash \textit{exists}\backslash\textit{forall}\ e :: \langle bool \rangle}$$

Conditional expressions :
$$\frac{\Gamma \vdash e_1 :: \langle bool \rangle \quad \Gamma \vdash e_2 :: t \quad \Gamma \vdash e_3 :: t}{\Gamma \vdash \textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ e_3\ \textit{endif} :: t}$$

Function and predicate definitions :
$$\frac{\Gamma, v1 :: \langle t_1 \rangle, \ldots, v_{n-1} :: \langle t_{n-1} \rangle \ \vdash e :: \langle t_n \rangle}{\Gamma \vdash function\ t_n : \mathbf{f}(t_1 : v_1, \ldots, t_{n-1} : v_{n-1}) = e}$$
$$\frac{\Gamma, v_1 :: \langle t_1 \rangle, \ldots, v_{n-1} :: \langle t_{n-1} \rangle \ \vdash e :: \langle bool \rangle}{\Gamma \vdash \textit{predicate} : \mathbf{p}(t_1 : v_1, \ldots, t_{n-1} : v_{n-1}) = e}$$

Function and predicate calls :
$$\frac{\Gamma \vdash f :: (t_1, ..., t_{n-1}) \rightarrow t_n \quad \Gamma \vdash e_i :: t'_i \quad t'_i <: t_i \vee o(t'_i) = t_i}{\Gamma \vdash f(e_1, \ldots, e_{n-1}) :: t_n}$$

Let expressions :
$$\frac{\Gamma_1 \vdash c \quad \Gamma \vdash e :: t}{\Gamma \vdash \textit{let}\ \{\mathsf{decls}\ c\}\ \textit{in}\ e :: t}$$

Constraints :
$$\frac{\Gamma \vdash e :: \langle var\ bool \rangle}{\Gamma \vdash constraint\ e}$$

Solve statements :

- Satisfaction statement:
$$\frac{}{\Gamma \vdash solve\ satisfy}$$
- Optimization statements:
$$\frac{\Gamma \vdash e :: t}{\Gamma \vdash solve\ minimize\backslash maximize\ e}$$

with o(t) as defined in Section 3.1.1 (p. 22) and type, vtype, genvar, cond and decls referring to the non-terminals of the grammar (Section 2.1.1 (p. 5)).

2.1.3 Semantics

In this subsection we present the semantic we have defined for the Constraint Programming language MiniZinc. This formalization allow us to state the completeness and soundness of our proposal.

MiniZinc models are characterized by their set of solutions. To establish the semantic interpretation of the language we define the evaluation of the expressions (Definition 2) with relation to well-typed substitutions (Definition 1) and which of this substitutions conforms a solution (Definition 3).

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* $\mathcal{M}$.[1]
- *For all* $x \in dom(\sigma)$, $\Gamma \vdash x ::< t >$ *iff* $\Gamma \vdash x\sigma ::< t >$

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 3.1.1 (p. 22) 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 $1 \leq \| e_2 \|_\sigma \leq n$.*

[1] 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.

6. *Set/list comprehensions of the form* $lc = \langle e \mid g_1, \ldots, g_m$ *where* $c\rangle$*, where:*
   - $\langle , \rangle$ *represents either {,} or [,]*
   - 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 its evaluation with respect to a substitution* α*:* $\| \mathcal{C}(e,c) \|_\alpha = \| \langle e \rangle \|_\alpha$ *if* $\| c \|_\alpha$ *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:*

   $\| \langle e\ | g \text{ where } c \rangle \|_\sigma =$
   $\| \, \| \mathcal{C}(e,c) \|_{id \mapsto e_1} \ \diamond \ldots \diamond \ \| \mathcal{C}(e,c) \|_{id \mapsto e_n} \|_\sigma \ =$ *(notation)*
   $\| \bigodot_{i=1}^{n} \left(\| \mathcal{C}(e,c) \|_{id \mapsto e_i} \right) \|_\sigma$

   (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:*

   $\| \langle e\ | g_1, \ldots, g_m \text{ where } c \rangle \|_\sigma =$
   $\| \, \| \langle e\ | g_2 \ldots, g_m \text{ where } c \rangle \|_{id \mapsto e_1} \ \diamond \ldots \diamond \ \| \langle e\ | g_2 \ldots, g_m \text{ where } c \rangle \|_{id \mapsto e_n} \|_\sigma$
   $=$ *(notation)*
   $\| \bigodot_{i=1}^{n} \left(\| \langle e\ | g_2 \ldots, g_m \text{ where } c \rangle \|_{id \mapsto e_i} \right) \|_\sigma$
   Notice that for list comprehension, the order of the generators is relevant for its evaluation.

7. *Set unions and list concatenations:*

   Let S_1*,* S_2 *be set or array expressions of the form* $S_1 = \langle s^1_1, \ldots, s^1_{n_1} \rangle$*,* $S_2 = \langle s^2_1, \ldots, s^2_{n_2} \rangle$*. Then:*

   $\| S_1 \diamond S_2 \|_\sigma = \| \langle s^1_1, \ldots, s^1_{n_1}, s^2_1, \ldots, s^2_{n_2} \rangle \|_\sigma$

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 (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 these 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, $\| c \|_\sigma = true$.*

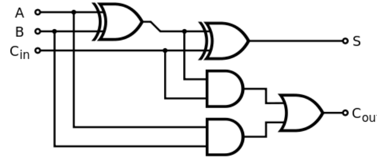

Figure 2.1: A n bit full adder circuit

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 $f\sigma' < f\sigma$)*

2.1.4 Examples

This section shows MiniZinc models for some of the problems our proposal deals with. First, the Combinational Circuit example models a full-adder using three valued logic; showing a real application [6] of Constraint Programming and extended types. Next, Time Optimization example shows a simple model to illustrate optimization problems in the language.

Combinational Circuit

The use of many valued logics in Constraint Programming is proposed by Azevedo [6] for testing and diagnosis of combinational circuits. This example is based in his proposal, modeling a n-bit adder with undefined (i.e. neither true or false) signals.

The basic piece of the circuit is the *full adder* (Figure 2.1) which adds binary numbers and accounts for values carried in as well as out.

A n-bit adder consists of n full adder modules. When considering Boolean types extended with an "undefined" value, the logic gates of the circuit behaves as indicated in Figure 2.4 (p. 18), where 0 and 1 stand for true and false, and $\perp$ stands for the extra "undefined" value.

In the model (Figures 2.2 (p. 16) 2.3 (p. 17)), the "extended bits" of the inputs, carries and output are modeled using for each of them two arrays, one representing the standard Boolean values and the other representing the additional "undef" value (following the ideas of the array transformation from Section 3.1 (p. 21)).

The code in Figure 2.2 (p. 16) define the input (xs, xe, ys, ye), output (ss, se) and carry (cs, ce) of the model and the auxiliary intermediate nodes of the circuit (aux*). Figure 2.3 (p. 17) shows the model constraints, where first

line states that the first carry is 0 and next line constraints the last output to be equal to the last carry. The following lines establish the constraints of the behavior of the circuit and last line structures the output of the model to be displayed as in the following example:

```
["1", "N", "0", "1"] + ["1", "N", "0", "0"] = ["0", "N", "N", "1", "0"]
c:["0", "1", "N", "0", "0"]
```

representing this solution:

$$
\begin{array}{cccccc}
x & = 1 & \bot & 0 & 1 & \\
y & = 1 & \bot & 0 & 0 & \\
c & = 0 & 1 & \bot & 0 & 0 \\
\hline
s & = 0 & \bot & \bot & 1 & 0
\end{array}
$$

Where 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 $\bot + \bot + 1$ (1 is the carry from the previous position) yields $\bot$ in the result. In particular this means that the carry is undefined as well, and thus in the third position $0+0+\bot$ produces the output $\bot$. However, in this case we can ensure that the carry is 0, and thus in the fourth position we have $1+0+0=1$ as output with 0 carry and as last bit.

Time Optimization

As an optimization example, we present a simple problem where an expression conformed by two additions and a simple constraint is minimized. The variables represents time, measured in hours, from 0 to 23, plus an especial value *oneDayOrMore*.

Notice that due to the time representation, if the sum of the two values exceeds 23 then it should be represented as *oneDayOrMore*.

The example in Figure 2.5 (p. 18) models an optimization problem, with the time representation mentioned above, where the addition of the time of two variables (*t1* and *t2*) and a parameter (t) have to be minimized, having that *t1* takes three hours more than *t2*.

In the example, the sum of the values of the parameters exceed 23 hours, and therefore even assuming the minimum possible value for $t1$ and $t2$ (which is 0), the expression takes the value *oneDayOrMore*. After transforming the model MiniZinc yields the expected values for variables *total*, *t1* and *t2*:

```
Total=oneDayOrMore  t1=3  t2=0
```

2.2 MiniZinc^{+}

This section describes MiniZinc^{+}, the language extension of MiniZinc we propose to allow the use of extended types.

2.2.1 Syntactic Extension

The grammar of MiniZinc+ is basically the grammar of MiniZinc adding only the possibility of declaring new, extended types:

typeE ⟶ extended tId =
[c_{-n}, ..., c_{-1}] ++type++ [c_1, ..., c_m]
type ⟶ int | bool | float | **tId** | range
model ⟶ **typeE**$^{*[;]}$;decl,$^{*[;]}$; assig$^{*[;]}$; pred$^{*[;]}$
; funct$^{*[;]}$;const$^{*[;]}$; solv; out;

The only difference of this grammar with respect to the standard MiniZinc with functions presented in [2] is the new non-terminal typeE and the inclusion of type identifiers (**tId**) as possible types.

2.2.2 Semantic Extension

As in previous section, the semantic of the language MiniZinc+ varies only w.r.t MiniZinc's one in the type system.

The syntactic extension allows extended type declarations, introducing new types. This types have an inferred total order for their values, defined following the textual order from the type declaration statement defined in Section 3.1.1 (p. 22). This order given to the new types is necessary for the definition of set semantics and for the optimization statements.

MiniZinc+ models also include a new predefined function (sv). This function receives an expression and returns true if the expression is a standard value and false otherwise.

2.2.3 Examples in MiniZinc+

This section shows how to model in MiniZinc+ language the examples of Section 2.1.4. We can observe in the following examples that our proposal simplifies the process.

Combinational Circuit

Section 2.1.4 shows a model which represents a n-bit adder with undefined values. This Section explains the MiniZinc+ model for the same problem from the new type and function definitions to the constrains used to model the circuit.

The definition of the new type can be found in the first line of the model in Figure 2.6 (p. 19). Note that replacing boolEx with bool in lines (3-6) and omitting lines (8-23) would give a standard MiniZinc model for this problem.

The model redefines the behavior of the Boolean connectives $\wedge$, $\vee$ and *xor* taking into account the new constant as indicated in the truth tables of Figure 2.4 (p. 18) (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 2.6 (p. 19). 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 2.4 (p. 18). The schema of this function will be usual in all the *conservative* redefinition of standard operators. The code for functions redefining $\wedge$ and $\vee$ is analogous.

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

Using these definitions, the code of lines (25-28) employs n full adders (Figure 2.1 (p. 12)) to model 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 presented in Section 2.1.4.

Time Optimization

The example described in Section 2.1.4 models an optimization problem for time variables with values $\{0, 1, \ldots, 23, \text{OneDayOrMore}\}$. This section explains how to model this problem in MiniZinc$^+$ language.

Figure 2.7 (p. 20) shows the code of the model. First, the new type is defined (line 1). Next, the variables of the model are declared. Then, addition operator $+$ is redefined, ensuring that if the sum of the two values exceeds 23 then the value *oneDayOrMore* is returned. Finally, the constraints of the model and the minimize and output statements are added.

```
1  int: n=4;
2
3  array [1..n] of var bool: xs;
4  array [1..n] of var bool: xe;
5  constraint forall([ xe[i] -> not(xs[i]) |i in 1..n]);
6
7  array [1..n] of var bool: ys;
8  array [1..n] of var bool: ye;
9  constraint forall([ ye[i] -> not(ys[i]) |i in 1..n]);
10
11 array [1..n+1] of var bool: ss;
12 array [1..n+1] of var bool: se;
13 constraint forall([ se[i] -> not(ss[i]) |i in 1..n+1]);
14
15 array [1..n+1] of var bool: cs;
16 array [1..n+1] of var bool: ce;
17 constraint forall([ ce[i] -> not(cs[i]) |i in 1..n+1]);
18
19 %% intermediate nodes:
20
21 %% a xor b:
22 array [1..n] of var bool: aux1s;
23 array [1..n] of var bool: aux1e;
24 constraint forall([ aux1e[i] -> not(aux1s[i]) |i in 1..n
       ]);
25
26 %% (a xor b) and c
27 array [1..n] of var bool: aux2s;
28 array [1..n] of var bool: aux2e;
29 constraint forall([ aux2e[i] -> not(aux2s[i]) |i in 1..n
       ]);
30
31 %% a and b
32 array [1..n] of var bool: aux3s;
33 array [1..n] of var bool: aux3e;
34 constraint forall([ aux3e[i] -> not(aux3s[i]) |i in 1..n
       ]);
```

Figure 2.2: A n bit adder in MINIZINC: $x + y = s$ (variable declarations)

```
1 constraint (cs[1] = false) /\ (ce[1] = false);
2 constraint (cs[n+1] = ss[n+1]) /\ (ce[n+1] =se[n+1]);
3
4 constraint forall ([ ((xe[i]=false) /\ (ye[i]=false)) -> ((
      aux1s[i] = (xs[i] xor ys[i])) /\ (aux1e[i] = false))  |i
      in 1..n]);
5
6 constraint forall ([ (not((xe[i]=false) /\ (ye[i]=false))) ->
      (aux1e[i] = true)  |i in 1..n]);
7
8 constraint forall ([ (((aux1e[i]=false) /\ (ce[i]=false)) \/
      ((aux1s[i] = false) /\ (aux1e[i] = false)) \/ ((cs[i] =
      false) /\ (ce[i]=false))) -> ((aux2s[i] = (aux1s[i] /\ cs[
      i])) /\ (aux2e[i] = false))  |i in 1..n]);
9
10 constraint forall ([ (not(((aux1e[i]=false) /\ (ce[i]=false))
      \/ ((aux1s[i] = false) /\ (aux1e[i] = false)) \/ ((cs[i] =
       false) /\ (ce[i]=false)))) -> (aux2e[i] = true)  |i in
      1..n]);
11
12 constraint forall ([ (((xe[i]=false) /\ (ye[i]=false)) \/ ((xs
      [i] = false) /\ (xe[i] = false)) \/ ((ys[i] = false) /\ (
      ye[i] = false))) -> ((aux3s[i] = (xs[i] /\ ys[i])) /\ (
      aux3e[i] = false))  |i in 1..n]);
13
14 constraint forall ([ (not(((xe[i]=false) /\ (ye[i]=false)) \/
      ((xs[i] = false) /\ (xe[i] = false)) \/ ((ys[i] = false)
      /\ (ye[i] = false)))) -> (aux3e[i] = true)  |i in 1..n]);
15
16 constraint forall ([ ((aux1e[i]=false) /\ (ce[i]=false)) -> ((
      ss[i] = (aux1s[i] xor cs[i])) /\ (se[i] = false))  |i in
      1..n]);
17
18 constraint forall ([ (not((aux1e[i]=false) /\ (ce[i]=false)))
      -> (se[i] = true)  |i in 1..n]);
19
20 constraint forall ([ (((aux2e[i]=false) \/ (aux3e[i]=false))
      \/ (aux2s[i] = true) \/ (aux3s[i] = true)) -> ((cs[i+1] =
      (aux2s[i] \/ aux3s[i]))/\ce[i+1]=false) | i in 1..n]);
21
22 constraint forall ([ (not(((aux2e[i]=false) \/ (aux3e[i]=false
      )) \/ (aux2s[i] = true) \/ (aux3s[i] = true))) -> (ce[i+1]
       = true) | i in 1..n]);
23
24 solve satisfy;
25
26 output([show([if fix(xe[i]) then "N" else show(bool2int(xs[i])
      ) endif | i in index_set(xs)])++" + " ++ show([if fix(ye[i
      ]) then "N" else show(bool2int(ys[i])) endif | i in
      index_set(ys)]) ++" = " ++ show([if fix(se[i]) then "N"
      else show(bool2int(ss[i])) endif | i in index_set(ss)]) ++
       " c:" ++ show([if fix(ce[i]) then "N" else show(bool2int(
      cs[i])) endif | i in index_set(cs)]) ++"\n"]);
```

Figure 2.3: A n bit adder in MiniZinc: $x + y = s$ (constraints)

	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

Figure 2.4: Truth tables including the undefined value

```
1  0..24: t=21;
2  var 0..24: t1;
3  var 0..24: t2;
4  var 0..24: total;
5
6  function var 0..24: add (var 0..24: a, var 0..24: b) =
7           let { var 0..24: r;
8           constraint (((a+b)>23) /\ (r = 24)) \/
9                      (((a+b)<=23) /\ (r = (a+b)))}
10         in r;
11
12 constraint  (t1 = add(3,t2));
13
14 constraint total = add(add(t1,t2),t);
15
16 solve minimize total;
17 output(["Total=",if (fix(total) = 24)
18                    then show("One_Day_Or_More")
19                    else show(total) endif,
20         " t1 =",if (fix(t1) = 24)
21                    then show("One_Day_Or_More")
22                    else show(t1) endif,
23         " t2 =", if (fix(t2) = 24)
24                    then show("One_Day_Or_More")
25                    else show(t2) endif,"\n"]);
```

Figure 2.5: Modeling time with an extended value..

```
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;
```

Figure 2.6: A n bit adder in MiniZinc+: $x + y = s$

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

Figure 2.7: Modelling time with an extended value..

Chapter 3

From MiniZinc$^+$ to MiniZinc

Previous chapter introduces the characteristics of MiniZinc$^+$, describing the differences with the source language MiniZinc. In this chapter we define and prove correct a program transformation from MiniZinc$^+$ to MiniZinc to compile the new language. Section 3.1 presents the ideas behind the transformation and how each language construction is transformed. In Section 3.2, the correctness of the transformation is proven to be correct by demonstrating the semantic equivalence of the original models and their transformations.

3.1 Transformation

This section presents a program transformation from MiniZinc$^+$ models to equivalent MiniZinc ones. Thanks to this translation, the models written in the extended setting can be solved using all the features (optimizations, different types of solvers, etc.) included in MiniZinc.

The translation can be presented as a process in two phases:

1. First, functions, predicates and local declarations of variables are removed from the model.

2. Finally, the resulting MiniZinc$^+$ model, now containing neither functions nor local declarations, is translated into MiniZinc.

Observe that the first phase can be applied to both MiniZinc and MiniZinc$^+$ indistinctly. In particular, the function elimination is done unrolling the function calls following ideas similar to those described in [2]. We assume in our setting the use of *total* and *pure* [2] functions, which simplifies the task. The elimination of constraints included in local declarations is managed using the relational semantics [10] 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 [11] for a more detailed discussion.

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

In the case of MiniZinc$^+$ expressions, we define two different transformations, the first one representing the standard MiniZinc part of the expression (transformation τ_s), and the second one keeping a representation of the extended part (transformation τ_e).

3.1.1 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 Section 2.1.1, 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 2.1.3.

3.1.2 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 2.6. 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.[1]

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 t is a standard type, then $\tau_s(e)$= $\{ \tau_s(e_1), \ldots, \tau_s(e_n) \}$, and $\tau_e(e)$=$\{\}$
- if t is a extended type, then

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

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

[1] 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.

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

3.1.3 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:

- 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 S where x=y].

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

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

Observe that during the evaluation of the model a will be evaluated to [ ] ++ [-1,8] ++ [posInf] = [-1,8,posInf]. The idea behind a is to obtain the list of elements in S without repetitions and respecting the order among elements. This mimics in MiniZinc$^+$ the behavior 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].

3.1.4 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.

3.1.5 Predefined function and predicate calls

We consider the following predefined function and predicate calls:

- $c \equiv sv(exp)$. The purpose of this Boolean function is to check if the expression exp corresponds to a standard value. Therefore: $\tau_s(c) = (\tau_e(exp) = z)$, with z the value zero value associated to the type of the expression.

- $c \equiv predef(f)(exp_1, \ldots, exp_n)$, 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 $\tau_e(c) = 0$.

- $c \equiv (S_1\ union\ S_2)$, assuming that *union* has not been redefined. Then: $\tau_s(c) = \tau_s(S_1)\ union\ \tau_s(S_2)$ and $\tau_e(c) = \tau_e(S_1)\ union\ \tau_e(S_2)$

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.

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

3.1.6 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.

3.1.7 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:

	$\mathcal{T}$
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, $type(setexp) = <\text{set of } t>$	
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}.

3.1.8 Assignments and Constraints

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

	$\mathcal{T}$
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).

3.1.9 Output Item

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

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

creates an array of string *tnames*

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

and replaces each show(e) by

if(fix^2($\tau_e(e)$)==0) then show($\tau_s(e)$) else show(tnames[$\tau_e(e)$]) endif

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

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

3.1.10 Optimization

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

MiniZinc also allows defining *optimization problems*, using solve minimize e or solve maximize e. In MiniZinc$^+$ we also allow the optimization of expressions with extended type, extending implicitly the order $<$ to the new elements accordingly to their position with respect to the standard type in the definition of the type extension (see Section 2.2.2).

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.

Let a, b be respectively the minimum and maximum value of the standard type if they exist. If a and/or b do not exist then we may be able to determine $a = \min(\tau_s(y))$ and $b = \max(\tau_s(y))$. As a last resort, if we are to use a solver which artificially represents unbounded objects of the base type in a finite range $a..b$ we can use these values. Note that most finite domain solvers have this restriction. If we cannot determine either a or b then the optimization cannot be translated.[3]

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

A time optimization problem can be found in Figure 2.7 . The time is measured in hours, from 0 to 23, plus an especial value *oneDayOrMore*. For this type $a = 0$ and $b = 23$.

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

3.2 Correctness

The transformation described in previous section translate a model written in MiniZinc$^+$ into a MiniZinc model. The soundness and correctness of this process is proven, demonstrating the semantic equivalence between the source code and its transformation.

The idea is to prove that both the MiniZinc$^+$ model and its transformation represent the same set of solutions.

Next definition establishes the transformation of the substitutions

Definition 4. *Let $\mathcal{M}$ be a* MiniZinc$^+$ *model and σ be a well-typed substitution for $\mathcal{M}$ (1), then,*

$$\begin{aligned}\sigma^{\mathcal{T}} = \; & \{\tau_s(x) \mapsto \tau_s(v) \mid (x \mapsto v) \in \sigma\} \cup \\ & \{\tau_e(x) \mapsto \tau_e(v) \mid (x \mapsto v) \in \sigma,\, \tau_e(x) \neq z_t\}\end{aligned}$$

We first introduce some useful technical results.

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

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

Proof. If e is a constant then both $\tau_s(e)$ and $\tau_e(e)$ are constants and Definition 2.2 shows that the evaluation w.r.t any substitution α behaves as the identity on constants. □

Lemma 2. *Let σ_1, σ_2 be disjoint substitutions, $\langle , \rangle$ and $\mathcal{C}(e, c)$ as in definition 2.6 then:*

$$\| \mathcal{C}(e, c) \|_{\sigma_1 \uplus \sigma_2} = \| \mathcal{C}(\| e \|_{\sigma_1}, \| c \|_{\sigma_1}) \|_{\sigma_2}$$

Proof. First we observe that σ_1 and σ_2 are disjoint substitutions and therefore, $\| c \|_{\sigma_1 \uplus \sigma_2} = \| \| c \|_{\sigma_1} \|_{\sigma_2}$.

Now we can prove the Lemma considering the following two cases:

1. If $\| c \|_{\sigma_1 \uplus \sigma_2}$ is $false$ then it is trivial to see that:

$$\| \mathcal{C}(e, c) \|_{\sigma_1 \uplus \sigma_2} = \| \mathcal{C}(\| e \|_{\sigma_1}, \| c \|_{\sigma_1}) \|_{\sigma_2} = \langle\rangle$$

2. If $\| c \|_{\sigma_1 \uplus \sigma_2}$ is $true$

$$\begin{aligned} & \| \mathcal{C}(e, c) \|_{\sigma_1 \uplus \sigma_2} = \\ & \text{By } \mathcal{C}(,) \text{ evaluation} \\ & \| \| \langle e \rangle \|_{\sigma_1} \|_{\sigma_2} = \\ & \text{By Definitions 2.4, 2.3a} \\ & \| \langle \| e \|_{\sigma_1} \rangle \|_{\sigma_2} = \end{aligned}$$

$$\text{By } \mathcal{C}(,) \text{ evaluation}$$
$$\|\, \mathcal{C}(\|\, e \,\|_{\sigma_1}, \|\, c \,\|_{\sigma_1}) \,\|_{\sigma_2}$$

□

Lemma 3. *Let* $\circ$ *be a* MiniZinc *relational operator,* k *a constant,* $\diamond$, $\langle,\rangle$ *and* $\mathcal{C}(e,c)$ *as in definition 2.6 then:*

$$\|\langle [a_1,\ldots,a_n][i] \mid i \textit{ in } 1..n \textit{ where } [b_1,\ldots,b_n][i] \circ k\rangle\|_\sigma = \|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}(a_j, b_j \circ k) \,\|_\emptyset\right) \|_\sigma$$

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.

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

By Definition 2.6a

$$\|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}([a_1,\ldots,a_n][i], [b_1,\ldots,b_n][i] \circ k) \,\|_{i\mapsto j}\right) \|_\sigma =$$

By Lemma 2, having $i \mapsto j = i \mapsto j \uplus \emptyset$

$$\|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}(\|\, [a_1,\ldots,a_n][i] \,\|_{i\mapsto j}, \|\, [b_1,\ldots,b_n][i] \circ k \,\|_{i\mapsto j}) \,\|_\emptyset\right) \|_\sigma =$$

By Definition 2.9

$$\|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}(\|\, [a_1,\ldots,a_n][i] \,\|_{i\mapsto j}, \|\, [b_1,\ldots,b_n][i] \,\|_{i\mapsto j} \circ \|\, k \,\|_{i\mapsto j}) \,\|_\emptyset\right) \|_\sigma =$$

By definition 2.5b

$$\|\bigodot_{j=1}^{n} \left(\|\mathcal{C}([\|\, a_1 \,\|_{i\mapsto j},\ldots,\|\, a_n \,\|_{i\mapsto j}][\|\, i \,\|_{i\mapsto j}], [\|\, b_1 \,\|_{i\mapsto j},\ldots,\|\, b_n \,\|_{i\mapsto j}][\|\, i \,\|_{i\mapsto j}] \circ \|\, k \,\|_{i\mapsto j})\|_\emptyset\right)\|_\sigma =$$

By Definitions 2.1 and 2.2, having that i does not appear in $[a_1,\ldots,a_n]$ or $[b_1,\ldots,b_n]$

$$\|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}([a_1,\ldots,a_n][j], [b_1,\ldots,b_n][j] \circ k) \,\|_\emptyset\right) \|_\sigma =$$

By array access

$$\|\bigodot_{j=1}^{n} \left(\|\, \mathcal{C}(a_j, b_j \circ k) \,\|_\emptyset\right) \|_\sigma$$

□

The soundness of the proposal will be a direct consequence of next Lemma and Lemma 9.

Lemma 4. *For every expression e of a simple type (i.e. e is neither set nor array expression) 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:

$$\text{There is some } v \text{ such that } e\sigma = v \text{ (Def. 1)} \tag{3.1}$$

Moreover, by Definition 4

$$e \mapsto v \in \sigma \text{ iff } \tau_s(e) \mapsto \tau_s(v) \in \sigma^\mathcal{T} \tag{3.2}$$

$$e \mapsto v \in \sigma, \tau_e(e) \neq z_t \text{ iff } \tau_e(e) \mapsto \tau_e(v) \in \sigma^\mathcal{T} \tag{3.3}$$

We distinguish two cases:

• If e is an standard type identifier.

$$\begin{aligned} \| \tau_s(\| e \|_\sigma) \|_\emptyset &= \text{(Def. 2.1)} = \| \tau_s(e\sigma) \|_\emptyset = \text{(Lemma 1)} = \\ &= \tau_s(e\sigma) = \text{(By (3.1))} = \underline{\tau_s(v)} \\ \| \tau_s(e) \|_{\sigma^\mathcal{T}} &= \text{(By (3.2))} = \underline{\tau_s(v)} \end{aligned}$$

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 $\tau_e(e\sigma) = z_t$

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

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

$$\begin{aligned} \| \tau_t(\| e \|_\sigma) \|_\emptyset &= \text{(Def. 2.1)} = \| \tau_t(e\sigma) \|_\emptyset = \text{(Lemma 1)} = \\ &= \tau_t(e\sigma) = \text{(By (3.1))} = \underline{\tau_t(v)} \\ \| \tau_t(e) \|_{\sigma^\mathcal{T}} &= \text{(Either by (3.2) if } t \text{ is } s \text{ or by (3.3) if } t \text{ is } e) = \underline{\tau_t(v)} \end{aligned}$$

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

- e is a constant:
  Let τ_t be either τ_s or τ_e.

$$\begin{aligned} \| \tau_t(\| e \|_\sigma) \|_\emptyset &= \text{(By Def. 2.2)} \\ \| \tau_t(e) \|_\emptyset &= \text{(By Lemma 1)} \\ \tau_t(e) &= \text{(By Lemma 1)} \\ \| \tau_t(e) \|_{\sigma^\mathcal{T}} \end{aligned}$$

- $e \equiv a[i]$ 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. 2.5a, $\| a[i] \|_\sigma = \| a \|_\sigma[\| i - (m-1) \|_\sigma]$. Let t be either s or e.

$$\begin{aligned} \| \tau_t(\| a[i] \|_\sigma) \|_\emptyset &= \text{(By Definition 2.5a)} \\ \| \tau_t(\| a \|_\sigma[\| i - (m-1) \|_\sigma]) \|_\emptyset &= \text{(By array access transformation)} \\ \| \tau_t(\| a \|_\sigma)[\tau_s(\| i - (m-1) \|_\sigma)] \|_\emptyset &= \text{(By Definition 2.5b,} \\ &\quad\ \text{having } \tau_t(\| a \|_\sigma) \text{ is an array literal.)} \\ \| \tau_t(\| a \|_\sigma) \|_\emptyset[\| \tau_s(\| i - (m-1) \|_\sigma) \|_\emptyset] &= \text{By structural induction} \\ \| \tau_t(a) \|_{\sigma^\mathcal{T}}[\| \tau_s(i - (m-1)) \|_{\sigma^\mathcal{T}}] &= \text{By Definition 2.5a} \\ \| \tau_t(a)[\tau_s(i)] \|_{\sigma^\mathcal{T}} &= \text{By array access transformation} \\ \| \tau_t(a[i]) \|_{\sigma^\mathcal{T}} \end{aligned}$$

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

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

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

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

$$\begin{aligned} \| \tau_e(\| forall(a) \|_\sigma) \|_\emptyset = \\ \| \tau_e(\| e_1 \|_\sigma \wedge \cdots \wedge \| e_n \|_\sigma) \|_\emptyset = \\ \| 0 \|_\emptyset \\ \| 0 \|_{\sigma^\mathcal{T}} = \\ \| \tau_e(forall(a)) \|_{\sigma^\mathcal{T}} = \\ \| \tau_e(e) \|_{\sigma^\mathcal{T}} \end{aligned} \tag{3.4}$$

Notice (3.4) 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.

□

To prove Lemma 9 we use the following auxiliary Lemmas:

Lemma 5. *Let σ_1, σ_2 be disjoint substitutions and S_1, S_2 sets:*
$\| S_1 \text{ union } S_2 \|_{\sigma_1 \uplus \sigma_2} = \| \| S_1 \|_{\sigma_1} \text{ union } \| S_2 \|_{\sigma_1} \|_{\sigma_2}$

Proof. Straightforward. □

Lemma 6. *Let α be a substitution, S_1, S_2 be sets or array expressions, τ_t be either τ_s or τ_e and $\diamond$ be the set union or array concatenation, then:*
$\| \tau_t(S_1 \ \diamond \ S_2) \|_\alpha = \| \tau_t(S_1) \ \diamond \ \tau_t(S_2) \|_\alpha$

Proof. Straightforward. □

Lemma 7. *Let exp, c an expression and a Boolean expression respectively, τ_t either τ_s or τ_e, α a substitution and $\mathcal{C}(exp, \tau_s(c))$ an array expression. Then:*
$\| \tau_t(\| \mathcal{C}(exp, \tau_s(c)) \|_\alpha) \|_\emptyset = \| \mathcal{C}(\tau_t(exp), \tau_s(c)) \|_{\alpha^\mathcal{T}}$

Proof. 1. If $\| c \|_\alpha$ is $false$, then $\| \tau_s(c) \|_\alpha$ is $false$ and therefore

$$\begin{aligned} \| \tau_t(\| \mathcal{C}(exp, \tau_s(c)) \|_\alpha) \|_\emptyset = \\ \text{By } \mathcal{C}(,) \text{ evaluation} \\ \| \tau_t(\| [] \|_\alpha) \|_\emptyset = \\ \text{By Definition 2.4 and array transformation} \\ [] = \\ \text{By Definition 2.4 and array transformation} \\ \| [] \|_{\alpha^\mathcal{T}} = \\ \text{By } \mathcal{C}(,) \text{ evaluation, having } \| \tau_s(c) \|_{\alpha^\mathcal{T}} = \text{false} \\ \| \mathcal{C}(\tau_t(exp), \tau_s(c)) \|_{\alpha^\mathcal{T}} \end{aligned}$$

2. If $\| c \|_\alpha$ is true, then:

By Lemma 1: $\| \tau_s(c) \|_\alpha = \| \tau_s(c) \|_{\alpha^\mathcal{T}} = true$

$$\begin{aligned}
&\| \tau_t(\| \mathcal{C}(exp, \tau_s(c)) \|_\alpha) \|_\emptyset = \\
&\text{By } \mathcal{C}(,) \text{ evaluation} \\
&\| \tau_t(\| [exp] \|_\alpha) \|_\emptyset = \\
&\text{By Definition 2.4 and array transformation} \\
&[\| \tau_t(\| exp \|_\alpha) \|_\emptyset] = \\
&\text{By Lemma 4} \\
&[\| \tau_t(exp) \|_{\alpha^\mathcal{T}}] = \\
&\text{Having that } \| \tau_s(c) \|_{\alpha^\mathcal{T}} = true \\
&\text{and by } \mathcal{C}(,) \text{ evaluation} \\
&\| \mathcal{C}(\tau_t(exp), \tau_s(c)) \|_{\alpha^\mathcal{T}}
\end{aligned}$$

□

Lemma 8. *Let exp, c an expression and a Boolean expression respectively, τ_t either τ_s or τ_e, α a substitution, $\circ_t$ be $=$ if τ_t stands for τ_s and $!=$ in other case and $\mathcal{C}(exp, \tau_s(c))$ be a set expression. Then:*
$\| \tau_t(\| \mathcal{C}(exp, \tau_s(c)) \|_\alpha) \|_\emptyset = \| \mathcal{C}(\tau_t(exp), \tau_s(c) \wedge \tau_e(exp) \circ_t 0) \|_{\alpha^\mathcal{T}}$

Proof.

1. If $\| c \|_\alpha$ is false, then:
   Similar to proof for this case in Lemma 7, having that if $\| \tau_s(c) \|_\alpha = false$ then $\| \tau_s(c) \wedge \tau_e(exp) \circ 0 \|_{\alpha^\mathcal{T}} = false$.

2. If $\| c \|_\alpha$ is true, then:

$$\begin{aligned}
&\| \tau_t(\| \mathcal{C}(exp, \tau_s(c)) \|_\alpha) \|_\emptyset = \\
&\| \text{By } \mathcal{C}(,) \text{ evaluation} \|_\emptyset \\
&\| \tau_t(\| \{exp\} \|_\alpha) \|_\emptyset = \\
&\text{By Definition 2.3a} \\
&\| \tau_t(\{\| exp \|_\alpha\}) \|_\emptyset = \\
&\text{By set transformation} \\
&\| \{[\tau_t(\| exp \|_\alpha)][i] \mid i \text{ in } 1..1 \text{ where } [\tau_e(\| exp \|_\alpha)][i] \circ_t 0\} \|_\emptyset = \\
&\text{By Lemma 3} \\
&\| \mathcal{C}(\tau_t(\| exp \|_\alpha), \tau_e(\| exp \|_\alpha) \circ_t 0) \|_\emptyset = \\
&\begin{cases} \| \{\tau_t(\| exp \|_\alpha)\} \|_\emptyset & \text{if } \| \tau_e(\| exp \|_\alpha) \circ_t 0 \|_\emptyset \\ \| \{\} \|_\emptyset & \text{otherwise} \end{cases} = \\
&\begin{cases} \underline{ord(\{\| \tau_t(\| exp \|_\alpha) \|_\emptyset\})} & \text{if } \| \tau_e(\| exp \|_\alpha) \circ_t 0 \|_\emptyset \\ \underline{\| \{\} \|_\emptyset} & \text{otherwise} \end{cases}
\end{aligned}$$

$$\| \mathcal{C}(\tau_t(e), \tau_s(c) \wedge \tau_e(e) \circ_t 0) \|_{\alpha^{\mathcal{T}}} =$$

Having $\| \tau_s(c) \|_{\alpha^{\mathcal{T}}} = true$ by Lemma 1

$$\| \mathcal{C}(\tau_t(e), \tau_e(e) \circ_t 0) \|_{\alpha^{\mathcal{T}}} =$$

$$\begin{cases} \| \{\tau_t(e)\} \|_{\alpha^{\mathcal{T}}} & \text{if } \| \tau_e(e) \circ_t 0 \|_{\alpha^{\mathcal{T}}} \\ \| \{\} \|_{\alpha^{\mathcal{T}}} & \text{otherwise} \end{cases} =$$

By Lemma 4, and Definitions 2.9, 2.2, 2.3a

$$\begin{cases} \underline{ord(\{\| \tau_t(\| e \|_\alpha) \|_\emptyset\})} & \text{if } \| \tau_e(\| e \|_\alpha) \circ_t 0 \|_\emptyset \\ \underline{\| \{\} \|_\emptyset} & \text{otherwise} \end{cases}$$

□

Lemma 9. *For every array or set 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.

e is a set expression

Let

$$ord(\{\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma\}) = \{\| e_{p_1} \|_\sigma, \ldots, \| e_{p_m} \|_\sigma\} \tag{3.5}$$

- 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 $\tau_e(e) = \{\}$

$$\| \tau_s(\| \{e_1, \ldots, e_n\} \|_\sigma) \|_\emptyset =$$

By Definition 2.3a

$$\| \tau_s(ord(\{\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma\})) \|_\emptyset =$$

By (3.5)

$$\| \tau_s(\{\| e_{p_1} \|_\sigma, \ldots, \| e_{p_m} \|_\sigma\}) \|_\emptyset =$$

By set transformation

$$\| \{\tau_s(\| e_{p_1} \|_\sigma), \ldots, \tau_s(\| e_{p_m} \|_\sigma)\} \|_\emptyset =$$

By Definition 2.3a

$$\underline{ord(\{\| \tau_s(\| e_{p_1} \|_\sigma) \|_\emptyset, \ldots, \| \tau_s(\| e_{p_m} \|_\sigma) \|_\emptyset\})} \tag{3.6}$$

$$\| \tau_s(\{e_1, \ldots, e_n\}) \|_{\sigma^{\mathcal{T}}} =$$

By set transformation

$$\| \{\tau_s(e_1), \ldots, \tau_s(e_n)\} \|_{\sigma\mathcal{T}} =$$
By Definition 2.3a
$$ord(\{\| \tau_s(e_1) \|_{\sigma\mathcal{T}}, \ldots, \| \tau_s(e_n) \|_{\sigma\mathcal{T}}\}) =$$
By Lemma 4
$$ord(\{\| \tau_s(\| e_1 \|_\sigma) \|_\emptyset, \ldots, \| \tau_s(\| e_n \|_\sigma) \|_\emptyset\}) =$$
By (3.5), repetition elimination and reordering
$$\underline{ord(\{\| \tau_s(\| e_{p_1} \|_\sigma) \|_\emptyset, \ldots, \| \tau_s(\| e_{p_m} \|_\sigma) \|_\emptyset\})} \tag{3.7}$$

Thus, by (3.6) = (3.7), $\| \tau_s(\| \{e_1, \ldots, e_n\} \|_\sigma) \|_\emptyset = \| \tau_s(\{e_1, \ldots, e_n\}) \|_{\sigma\mathcal{T}}$

$$\| \tau_e(\| e \|_\sigma) \|_\emptyset =$$
By Definition 2.3a
$$\| \tau_e(ord(\{\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma\})) \|_\emptyset =$$
By set transformation
$$\| ord(\{\}) \|_\emptyset =$$
$$\| \{\} \|_\emptyset =$$
By Definition 2.3a
$$ord(\{\}) = \tag{3.8}$$
By Definition 2.3a
$$\| \{\} \|_{\sigma\mathcal{T}} = \tag{3.9}$$
By set transformation
$$\| \tau_e(e) \|_{\sigma\mathcal{T}}$$

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

$$\| \tau_s(\| e \|_\sigma) \|_\emptyset =$$
$$\| \tau_s(ord(\{\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma\})) \|_\emptyset =$$
By (3.5)
$$\| \tau_s(\{\| e_{p_1} \|_\sigma, \ldots, \| e_{p_m} \|_\sigma\}) \|_\emptyset =$$
By set transformation
$$\|\{[\tau_s(\| e_{p_1} \|_\sigma), \ldots, \tau_s(\| e_{p_m} \|_\sigma)][i] \ \mid \ i \text{ in } 1..m$$
$$\text{where } [\tau_e(\| e_{p_1} \|_\sigma), \ldots, \tau_e(\| e_{p_m} \|_\sigma)][i] = 0\}\|_\emptyset =$$
By Lemma 3
$$\| \bigodot_{j=1}^{m} \big(\| \mathcal{C}(\tau_s(\| e_{p_j} \|_\sigma), \tau_e(\| e_{p_j} \|_\sigma) = 0) \|_\emptyset\big) \|_\emptyset =$$
By Lemma 2, having $\emptyset = \emptyset \uplus \emptyset$
$$\| \bigodot_{j=1}^{m} \big(\| \mathcal{C}(\| \tau_s(\| e_{p_j} \|_\sigma) \|_\emptyset, \| \tau_e(\| e_{p_j} \|_\sigma) = 0 \|_\emptyset) \|_\emptyset\big) \|_\emptyset =$$

$$\begin{array}{r}\text{By Definitions 2.2, 2.9}\\ \| \bigodot_{j=1}^{m} \left(\| \mathcal{C}(\| \tau_s(\| e_{p_j} \|_\sigma) \|_\emptyset, \| \tau_e(\| e_{p_j} \|_\sigma) \|_\emptyset = 0) \|_\emptyset\right) \|_\emptyset = \\ \text{By Lemma 4}\\ \| \bigodot_{j=1}^{m} \left(\| \mathcal{C}(\| \tau_s(e_{p_j}) \|_{\sigma^\mathcal{T}}, \| \tau_e(e_{p_j}) \|_{\sigma^\mathcal{T}} = 0) \|_\emptyset\right) \|_\emptyset\end{array}$$

$$\begin{array}{r}\| \tau_s(e) \|_{\sigma^\mathcal{T}} = \\ \text{By set transformation}\\ \|\{[\tau_s(e_1), \ldots, \tau_s(e_n)][i] \ \mid \ i \ in \ 1..n \\ \text{where } [\tau_e(e_1), \ldots, \tau_e(e_n)][i] = 0\}\|_{\sigma^\mathcal{T}} = \\ \text{By Lemma 3}\\ \| \bigodot_{j=1}^{n} \left(\| \mathcal{C}(\tau_s(e_j), \tau_e(e_j) = 0) \|_\emptyset\right) \|_{\sigma^\mathcal{T}} = \\ \text{By Lemma 5}\\ \| \bigodot_{j=1}^{n} \left(\| \mathcal{C}(\tau_s(e_j), \tau_e(e_j) = 0) \|_{\sigma^\mathcal{T}}\right) \|_\emptyset = \\ \text{By Lemma 2}\\ \| \bigodot_{j=1}^{n} \left(\| \mathcal{C}(\| \tau_s(e_j) \|_{\sigma^\mathcal{T}}, \| \tau_e(e_j) = 0 \|_{\sigma^\mathcal{T}}) \|_\emptyset\right) \|_\emptyset = \\ \text{By Definitions 2.2, 2.9}\\ \| \bigodot_{j=1}^{n} \left(\| \mathcal{C}(\| \tau_s(e_j) \|_{\sigma^\mathcal{T}}, \| \tau_e(e_j) \|_{\sigma^\mathcal{T}=0}) \|_\emptyset\right) \|_\emptyset = \\ \text{By repetition elimination using (3.5)}\\ \| \bigodot_{j=1}^{m} \left(\| \mathcal{C}(\| \tau_s(e_{p_j}) \|_{\sigma^\mathcal{T}}, \| \tau_e(e_{p_j}) \|_{\sigma^\mathcal{T}} = 0) \|_\emptyset\right) \|_\emptyset\end{array} \tag{3.10}$$

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

$$\begin{array}{r}\| \tau_t(\| e \|_\sigma) \|_\emptyset = \\ \| \tau_t(\| [e_1, \ldots, e_n] \|_\sigma) \|_\emptyset = \\ \text{By Definition 2.4}\\ \| \tau_t([\| e_1 \|_\sigma, \ldots, \| e_n \|_\sigma]) \|_\emptyset =\end{array}$$

By array transformation

$$\| [\tau_t(\| e_1 \|_\sigma), \ldots, \tau_t(\| e_n \|_\sigma)] \|_\emptyset =$$

By Definition 2.4

$$[\| \tau_t(\| e_1 \|_\sigma) \|_\emptyset, \ldots, \| \tau_t(\| e_n \|_\sigma) \|_\emptyset] =$$

By Lemma 4

$$\| \tau_t(e_1) \|_{\sigma^\tau}, \ldots, \| \tau_t(e_n) \|_{\sigma^\tau} =$$

By Definition 2.4

$$\| [\tau_t(e_1), \ldots, \tau_t(e_n)] \|_{\sigma^\tau} =$$

By array transformation

$$\| \tau_t([e_1, \ldots, e_n]) \|_{\sigma^\tau} =$$

$$\| \tau_t(e) \|_{\sigma^\tau}$$

- e is a set/list comprehension of the form $\langle exp \mid G_1, \ldots, G_m$ 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 t be either s or e and $\circ_t$ be $=$ if t is s or $! =$ if t is e. Let $\tau_g(c)$ be:
    * $\tau_s(c)$ if e is a list comprehension
    * $\tau_s(c) \wedge \tau_e(exp) \circ_t 0$ if e is a set comprehension

    By comprehension expression transformation $\tau_t(\langle exp \mid G_m$ where $cond\rangle) = \langle \tau_t(exp') \mid G'_m$ where $\tau_g(cond')\rangle$ where:
    * If ge_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 ge_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 $[ord_t^{-1}(\mathsf{x}) \mid \mathsf{x}$ in $\tau_e(ge_m)$ where x<0]++[x | x in $\tau_s(ge_m)$]++$[ord_t^{-1}(\mathsf{x})$ | x in $\tau_e(ge_m)$ where 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 occurrence of g_m has been changed by the array access $a[f]$.

    1. 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:

$$e_1 \prec e_2 \prec \ldots \prec e_n \tag{3.11}$$

$$\begin{aligned}
&\exists i,j \in 1..n \mid i \le j \wedge \\
&\qquad (\forall k \in 1..i-1\ \tau_e(e_k) < 0 \wedge \\
&\qquad \forall k \in i..j-1\ \tau_e(e_k) = 0 \wedge \\
&\qquad \forall k \in i..n\ \tau_e(e_k) > 0) && (3.12)\\
&\qquad \wedge\ ord(\tau_e(\{\tau_e(e_1),\ldots,\tau_e(e_{i-1})\})) = \\
&\qquad \{\tau_e(e_1),\ldots,\tau_e(e_{i-1})\} && (3.13)\\
&\qquad \wedge\ ord(\tau_e(\{\tau_s(e_i),\ldots,\tau_s(e_{j-1})\})) = \\
&\qquad \{\tau_s(e_i),\ldots,\tau_s(e_{j-1})\} && (3.14)\\
&\qquad \wedge\ ord(\tau_e(\{\tau_s(e_j),\ldots,\tau_s(e_n)\})) = \\
&\qquad \{\tau_e(e_j),\ldots,\tau_e(e_n)\} && (3.15)
\end{aligned}$$

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 $\|\, \tau_e(ge_m) \,\|_{\sigma^\tau}$:

Let t be either s or e, then:

$$\begin{aligned}
&\|\, \tau_t(ge_m) \,\|_{\sigma^\tau} = \\
&\text{By structural induction}\\
&\|\, \tau_t(\|\, ge_m \,\|_\sigma) \,\|_\emptyset = \\
&\|\, \tau_t(\{e_1, \ldots, e_m\}) \,\|_\emptyset = \\
&\text{By set transformation}\\
&\|\{[\tau_t(e_1), \ldots, \tau_t(e_n)][i] \mid i \text{ in } 1..n \text{ where}\\
&\qquad [\tau_e(e_1), \ldots, \tau_e(e_n)][i] \circ_t 0\}\|_\emptyset = \\
&\text{By Lemma (3)}\\
&\|\, \bigodot_{l=1}^{n} \left(\|\, \mathcal{C}(\tau_t(e_l), \tau_e(e_l) \circ_t 0) \,\|_\emptyset\right) \,\|_\emptyset = \\
&\text{By (3.12)}\\
&= \begin{cases} \|\, \bigodot_{i \le l < j} \left(\|\, \tau_s(e_l) \,\|_\emptyset\right) \,\|_\emptyset & \text{if } t \text{ stands for } s \\ \|\, \bigodot_{\substack{1 \le l < i \\ j \le l \le n}} \left(\|\, \tau_e(e_l) \,\|_\emptyset\right) \|_\emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Previous expressions are of the form $\|\, \bigodot_{l=i_a}^{i_b} \left(\|\, \tau_t(e_l) \,\|_\emptyset\right) \,\|_\emptyset$ and can be evaluated further:

$$\| \bigodot_{l=i_a}^{i_b} (\| \tau_t(e_l) \|_\emptyset) \|_\emptyset =$$

$$ord(\{\| \tau_t(e_{i_a}) \|_\emptyset, \ldots, \| \tau_t(e_{i_b}) \|_\emptyset\}) =$$

By Definition 2.7 and (3.13), (3.14), (3.15)

$$= \begin{cases} \{\tau_s(e_i), \ldots, \tau_s(e_{j-1})\} & \text{if } t \text{ stands for } s \\ \{\tau_t(e_1), \ldots, \tau_t(e_{i-1}), \tau_s(e_j), \ldots, \tau_s(e_n)\} & \text{otherwise} \end{cases}$$

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

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

   $$\| [x | x \text{ in } \tau_s(ge_m)] \|_{\sigma\tau} =$$

   Is syntax sugar of:

   $$\| [x | x \text{ in } \tau_s(ge_m) \text{ where } true] \|_{\sigma\tau} =$$

   By previous $\tau_s(ge_m)$ evaluation and Definition 2.6a

   $$\| \bigodot_{l=i}^{j-1} (\| \mathcal{C}(\tau_s(x), true) \|_{x \mapsto e_l}) \|_{\sigma\tau} \quad (3.16)$$

   By $\mathcal{C}(,)$ definition

   $$\| \bigodot_{l=i}^{j-1} (\| \tau_s(e_l) \|_\emptyset) \|_{\sigma\tau} =$$

   By Definition 2.7

   $$[\| \tau_s(e_i) \|_{\sigma\tau}, \ldots, \| \tau_s(e_{j-1}) \|_{\sigma\tau}] =$$

   By constant transformation, having

   $(\tau_e(e_i) = z_t, \ldots, \tau_e(e_{j-1}) = z_t$ by (3.12))

   $$[\| e_i \|_{\sigma\tau}, \ldots, \| e_{j-1} \|_{\sigma\tau}] =$$

   By Definition 2.4

   $$\| [e_i, \ldots, e_{j-1}] \|_{\sigma\tau}$$

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

   $$\| [ord_t^{-1}(x) | x \text{ in } \tau_e(ge_m) \text{ where } x < 0] \|_{\sigma\tau} =$$

   By previous $\tau_e(ge_m)$ evaluation and 2.6a

   $$\| \bigodot_{\substack{1 \le l < i \\ j \le l \le n}} (\| \mathcal{C}(ord_t^{-1}(x), x < 0) \|_{x \mapsto \tau_e(e_l)}) \|_{\sigma\tau} =$$

By Definitions 2.2, 2.9
(Applied like in Lemma 3 demo)

$$\| \bigodot_{\substack{1\leq l<i\\ j\leq l\leq n}} \left(\| \mathcal{C}(ord_t^{-1}(\tau_e(e_l)), \tau_e(e_l) < 0) \|_\emptyset\right) \|_{\sigma^\mathcal{T}} =$$

By $\mathcal{C}(,)$ evaluation and (3.13),(3.15)

$$\| \bigodot_{1\leq l<i} \left(\| ord_t^{-1}(\tau_e(e_l)) \|_\emptyset\right) \|_{\sigma^\mathcal{T}} =$$

By Definition 2.2

$$\| \bigodot_{1\leq l<i} \left(ord_t(ord_t^{-1}(\tau_e(e_l)))\right) \|_{\sigma^\mathcal{T}} =$$

By Definition 2.7

$$\|[e_1, \ldots, e_{i-1}]\|_{\sigma^\mathcal{T}}$$

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

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

$$\| x \|_{x\mapsto e_j} = \| a[f] \|_{f\mapsto j} \tag{3.17}$$

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

$$\| \tau_s(\| e \|_\sigma) \|_\emptyset =$$

$$\| \tau_s(\| \langle exp \; |G_m \text{ where } cond\rangle \|_\sigma) \|_\emptyset =$$

By Definition 2.6a

$$\| \tau_s(\| \bigodot_{i=1}^{n} \left(\| \mathcal{C}(exp, cond) \|_{gm\mapsto e_i}\right) \|_\sigma) \|_\emptyset =$$

By structural induction

$$\| \tau_s(\bigodot_{i=1}^{n} \left(\| \mathcal{C}(exp, cond) \|_{gm\mapsto e_i}\right)) \|_{\sigma^\mathcal{T}} =$$

By Lemma 6

$$\| \bigodot_{i=1}^{n} \left(\tau_s(\| \mathcal{C}(exp, cond) \|_{gm\mapsto e_i})\right) \|_{\sigma^\mathcal{T}} =$$

By (3.17)

$$\| \bigodot_{i=1}^{n} \left(\tau_s(\| \mathcal{C}(exp', cond') \|_{f\mapsto i})\right) \|_{\sigma^\mathcal{T}} =$$

By Lemma 7 or 8,
having by Definition 4 that:
$f \mapsto i^{\mathcal{T}}$ is $f \mapsto i$ (since f is an standard type identifier)

$$\| \bigodot_{i=1}^{n} \left(\| \mathcal{C}(\tau_s(exp'), \tau_g(cond')) \|_{f \mapsto i} \right) \|_{\sigma^{\mathcal{T}}} =$$

By Definition 2.6a

$$\| \langle \tau_s(exp') \mid g_m \text{ in } G'_m \text{ where } \tau_g(cond') \rangle \|_{\sigma^{\mathcal{T}}} =$$
$$\| \tau_s(\langle exp \mid G_m \text{ where } cond \rangle) \|_{\sigma^{\mathcal{T}}} =$$
$$\| \tau_s(e) \|_{\sigma^{\mathcal{T}}}$$

– 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 $\| ge_1 \|_\sigma = \langle e_1, \ldots, e_n \rangle$:

$$\| \tau_s(\| e \|_\sigma) \|_\emptyset =$$
$$\| \tau_s(\| \langle exp \mid G_1, \ldots, G_m \text{ where } cond \rangle \|_\sigma) \|_\emptyset =$$

By Definition 2.6b

$$\| \tau_s(\| \bigodot_{i=1}^{n} \left(\| \langle exp \mid G_2 \ldots, G_m \text{ where } cond \rangle \|_{g_1 \mapsto e_i} \right) \|_\sigma) \|_\emptyset =$$

By structural induction

$$\| \tau_s(\bigodot_{i=1}^{n} (\| \langle exp \mid G_2 \ldots, G_m \text{ where } cond \rangle \|_{g_1 \mapsto e_i})) \|_{\sigma^{\mathcal{T}}} =$$

By Lemma 6

$$\| \bigodot_{i=1}^{n} \left(\tau_s(\| \langle exp \mid G_2 \ldots, G_m \text{ where } cond \rangle \|_{g_1 \mapsto e_i}) \right) \|_{\sigma^{\mathcal{T}}} =$$

By definition2.6b

$$\| \tau_s(\langle exp \mid G_1, \ldots, G_m \text{ where } cond \rangle) \|_{\sigma^{\mathcal{T}}} =$$
$$\| \tau_s(e) \|_{\sigma^{\mathcal{T}}}$$

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

□

Corollary 1. *For every e and well-typed substitution σ:*

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

Proof. Straightforward from Lemmas 4 and 9 □

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. 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 $id = \tau_s(e)$.

$$
\begin{aligned}
\| a \|_\sigma = true &\Rightarrow \\
\| id = e \|_\sigma = true &\Rightarrow \text{By Definition 2.8} \\
\| id \|_\sigma = \| e \|_\sigma &\Rightarrow \text{Applying } \| \tau_s() \|_\emptyset \text{ both sizes} \\
\| \tau_s(\| id \|_\sigma) \|_\emptyset = \| \tau_s(\| e \|_\sigma) \|_\emptyset &\Rightarrow \text{By Corollary 1} \\
\| \tau_s(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By transformation} \\
\| id \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2.8} \\
\| id = \tau_s(e) \|_{\sigma^{\mathcal{T}}} = true &
\end{aligned}
$$

   - 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.

$$
\begin{aligned}
\| a \|_\sigma = true &\Rightarrow \\
\| id = e \|_\sigma = true &\Rightarrow \text{By Definition 2.8} \\
\| id \|_\sigma = \| e \|_\sigma &\Rightarrow \text{Applying } \| \tau_t() \|_\emptyset \text{ both sizes} \\
\| \tau_t(\| id \|_\sigma) \|_\emptyset = \| \tau_t(\| e \|_\sigma) \|_\emptyset &\Rightarrow \text{By Corollary 1} \\
\| \tau_t(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_t(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2.8} \\
\| \tau_t(id) = \tau_t(e) \|_{\sigma^{\mathcal{T}}} = true &
\end{aligned}
$$

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 $id = \tau_s(e)$.

$$
\| a' \|_{\sigma^{\mathcal{T}}} = true \Rightarrow
$$

$$\begin{aligned}
\| id = \tau_s(e) \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \text{By Definition 2.8} \\
\| id \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By identifier transformation} \\
\| \tau_s(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2. 1} \\
\tau_s(id)\sigma^{\mathcal{T}} = \| \tau_s(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Corollary 1} \\
\tau_s(id)\sigma^{\mathcal{T}} = \| \tau_s(\| e \|_{\sigma}) \|_{\emptyset} &\Rightarrow \\
\tau_s(id)\sigma^{\mathcal{T}} = \tau_s(\| e \|_{\sigma}) &\Rightarrow \\
\tau_s(id) \mapsto \tau_s(\| e \|_{\sigma}) \in \sigma^{\mathcal{T}} &\Rightarrow \text{By Definition 4} \\
id \mapsto \| e \|_{\sigma} \in \sigma &\Rightarrow \\
\| id \|_{\sigma} = \| e \|_{\sigma} &\Rightarrow \text{By Definition 2.8} \\
\| id = e \|_{\sigma} = true &
\end{aligned}$$

- 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.

$$\begin{aligned}
\| a' \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \\
\| \tau_t(id) = \tau_t(e) \|_{\sigma^{\mathcal{T}}} = true &\Rightarrow \text{By Definition 2.8} \\
\| \tau_t(id) \|_{\sigma^{\mathcal{T}}} = \| \tau_t(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Definition 2.1} \\
\tau_t(id)\sigma^{\mathcal{T}} = \| \tau_t(e) \|_{\sigma^{\mathcal{T}}} &\Rightarrow \text{By Corollary 1} \\
\tau_t(id)\sigma^{\mathcal{T}} = \| \tau_t(\| e \|_{\sigma}) \|_{\emptyset} &\Rightarrow \\
\tau_t(id)\sigma^{\mathcal{T}} = \tau_t(\| e \|_{\sigma}) &\Rightarrow \\
\tau_t(id) \mapsto \tau_t(\| e \|_{\sigma}) \in \sigma^{\mathcal{T}} &\Rightarrow \text{By Definition 4} \\
id \mapsto \| e \|_{\sigma} \in \sigma &\Rightarrow \\
\| id \|_{\sigma} = \| e \|_{\sigma} &\Rightarrow \text{By Definition 2.8} \\
\| id = e \|_{\sigma} = true &
\end{aligned}$$

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 Corollary 1, $\| \tau_s(c) \|_{\sigma^{\mathcal{T}}} = true$

3.- If S is of the form *maximize* f (respectively *minimize* f) then there is no well-typed substitution θ for $\mathcal{M}$ verifying 1) and 2) and such that $f\theta > f\sigma$ (respectively $f\theta < f\sigma$)

Analogously, item 3 requires a generalization of the following result: *For every pair of constants k, k' of some type t in $\mathcal{M}$ $k \leq k'$ (with the order $<$ extended to the new types in Section 2.2.2 (p. 14)) iff*

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

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

This result shows that if a substitution σ maximizes/minimizes the optimization statement of $\mathcal{M}$ then $\sigma^{\mathcal{T}}$ maximizes/minimizes the transformation of the optimization statement in $\mathcal{M}^{\mathcal{T}}$. $\square$

Chapter 4

Prototype

The ideas presented in this work have been implemented in a working prototype. This chapter shows the functionalities of the tool (Section 4.1), the architecture of the implementation (Section 4.2), explains how to install and use the prototype (Section 4.3) and finishes presenting the conclusions obtained from the development and the future work.

A detailed evaluation of the performance can be found in the Experimental Results section in Chapter 5.

4.1 Functionalities

This Section presents the characteristics of the current version of the prototype, explaining its main functionalities.

In Section 3.1, we state that the transformation process consists of two phases. First, functions, predicates and local declarations of variables are removed from the model. Then, the resulting MiniZinc$^+$ model without functions and local declarations is translated into MiniZinc.

We have implemented the second phase of the process as explained in Section 3.1, however, the already existing tools for function and local declaration elimination [2] do not perform the transformation for extended types. As an alternative, we have implemented a MiniZinc$^+$ to MiniZinc function and local declaration transformation:

First phase A extended function is translated into two translated functions, and their parameters transformed in their respective standard and extended transformation. Let statements inside function definitions of the form
$c \equiv let\ \{d_1, \ldots, d_n,\ c_1, \ldots, c_m\}\ in\ e$
are directly translated as:
$\tau_t(c) = let\ \{{d_1}^{\mathcal{T}}, \ldots, {d_n}^{\mathcal{T}},\ {c_1}^{\mathcal{T}}, \ldots, {c_m}^{\mathcal{T}}\}\ in\ \tau_t(e)$

This transformation is based on the totality and purity of the allowed MiniZinc+, forcing that each local variable can only take one value for each set of parameters.

Second phase The project is focused in this part of the process, detailing the transformation of each of the language constructions. The current version of the prototype implements all the transformations needed for dealing with simple extended types, however, the transformations of the data structures such as sets or arrays have not been developed yet.

4.2 Architecture and Implementation

The tool consist of two different modules: the parser and the MiniZinc+ Expression Manager.

The parser is the module that translates a MiniZinc+ model into its representation in the MiniZinc+ Expression Manager. It consists of a lexical and a syntactic analyzer for the proposed syntax (Section 2.1.3). They are developed using Flex [13] and Bison [14].

The MiniZinc+ Expression Manager contains the data modeling of the MiniZinc+ and MiniZinc expressions and implements the transformation process.

4.3 Manual

This section shows how to install and use the prototype. The instructions are detailed for a Linux system.

Dependencies To compile and run the prototype you should have installed the following software:

Mandatory software	Optional software
bison++	scmzn2mzn
flex++	fz (Gecode FlatZinc solver)
g++	STCG
MiniZinc	

Download To download current version download the file http://gpd.sip.ucm.es/rafa/minizinc/extendedMiniZinc.tar.gz.

Compilation To compile the prototype:

1. Extract the content from the downloaded archive file:
   tar -xvf extendedMiniZinc.tar.gz

2. Change directory to source folder:
   cd extendedMiniZinc

3. Build the software:
   make

Execution

1. Compile the MiniZinc+ model "inputFile" to MiniZinc+ with functions model "outputFile":
   parser inputFile 1> outputFile

2. Compile the MiniZinc with functions code to standard MiniZinc code:[1]
   chmod +x script_for_scmzn2mzn
   ./script_for_scmzn2mzn inputFile outputFile

3. outputFile.pr.mzn can be now executed as MiniZinc model with the following command:
   minizinc outputFile.pr.mzn

4.4 Conclusions and future development

The prototype shows the feasibility of the proposal and has been successfully applied to the problems that motivated the work. The length issues observed in the experimental results (Section 5.3 (p. 54)) for large models can be improved by performing Common Subexpression Elimination, enhancing the performance of the proposal [2].

The implementation of the transformation for set and array terms will complete the functionalities of the prototype.

[1]The shell script 'script_for_scmzn2mzn' will only work in Linux (and maybe Unix like) systems.

Chapter 5

Practical Application: SQL Test Case Generation

This chapter presents a practical application of MiniZinc$^+$. As introduced in the motivation of the work, the project was originally motivated by the need of using NULL values in the SQL Constraint Programming models generated by *STCG* [15]. This problem is used now to illustrate the feasibility and appropriateness of our proposal.

Section 5.1 introduces *STCG* tool and the models it generates. Next, the MiniZinc$^+$ code needed to allow NULL values for this models is explained in Section 5.2. Then experimental results obtained from the studied models are analyzed in Section 5.3.

5.1 SQL Test Case Generator

The tool *STCG* [15] generates MiniZinc models whose solutions constitute test cases for testing SQL views. Although realistic test-cases involve generating values for tables with several rows and queries relating different SQL views, we show here a very simple case of a test case for a SQL view defined as

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

with T a table defined as

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

Observe that the condition indicates that the expression (a != b or a != c) must be evaluated to NULL. The disjunction is evaluated to NULL if one of its operands is NULL and the other is false or if both operands are NULL. The distinct operator is evaluated to NULL if one or more of its operands are

NULL. In fact NULL values are an important feature in the relational database model [16].

A Positive Test Case for SQL is defined as “a non-empty database instance such that the query/relation produces a non-empty result” [15]. Thus, for our example, following instance of the database conforms a Positive Test Case:

T		
a	*b*	*c*
0	NULL	0

5.2 Extending the model with NULL values.

In order to generate a model that can represent NULL values, we extend the models including two new types: [1]

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

The models include the definitions of the functions (=,!=,∨, ∧) for integer and Boolean types extended with *NULL* value. ∨ and ∧ functions are equivalents to the ∨ and ∧ functions for the circuit example 2.2.3, = and != are defined as follows:

```
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( /\ ) 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;
```

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

Observe the use of the built-in function *eq* representing the syntactic equality between two constants and the use of *predef* to call the not redefined version of a MiniZinc function or operand. For instance, in our example NULL=NULL is evaluated to NULL due to the redefinition of =, while eq(NULL,NULL) is evaluated to true.

In order to check the efficiency of the proposal we have tried two different examples of models produced by *STCG*:

1. Model *Sql Or* is 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"];
```

   The transformation of this code can be found in Appendix B.

2. Model *Board* represents the possible data in a more involved SQL database example presented in [15]:

```
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
```

Model	*Sql Or*	*Sql Or*$^+$	*Board*	*Board*$^+$
var. decl.	3	6	8	16
var. flat.	5	99	54	2032
funct. calls	4	254	419	374678
size (KB)	0.5	5.0	13.2	15946.2
transf. time		17		2923
solve time	0.50	0.29	0.32	2.21

Table 5.1: Experimental data for two models generated by STCG

```
 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 MiniZinc$^+$ code in Appendix A:

5.3 Experimental Results

Table 5.1 shows the data obtained with our current implementation. The two models in MiniZinc produced by *STCG* are called *Sql Or* and *Board*. The MiniZinc models obtained after introducing the new type, redefining the operators and applying the transformation previously described are called in the table *Sql Or*$^+$ and *Board*$^+$ (See previous section for a detailed description of the models).

The rows of the table contain:

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

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

- *funct. calls*: The number of function calls included in the code, including calls to the predefined operators $\{=,!=,\vee,\wedge\}$. For instance in the first example *STCG* generates a model including only 4 calls. After extending

the model to MiniZinc$^+$ to support NULLs and applying the transformation to obtain the equivalent MiniZinc model we obtain a model with 254 function calls.

- *size*: The size in Kbytes of the models. It can be seen that the size increases dramatically after the transformation.
- *transf. time*: Time required by the transformation in milliseconds. In the more complex example of *Board* about 3 seconds are required by our prototype to convert the MiniZinc$^+$ model into a MiniZinc model.
- *solve time*: The time required by the MiniZinc solver to obtain the first answer in milliseconds[2].

The use of Common Subexpression Elimination is considered future work and is not implemented in the current version of the tool. This not only affects the solving performance [2] but also the number of function calls and the size of the model. Despite this increment in size, number of variables and function calls, the experimental results show that the theoretical proposal can be used in practice.

[2]Data from Gecode[17] FlatZinc solver statistical information

Chapter 6

Conclusions and future work

6.1 Conclusions

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 modeled using many-valued logics [18], or scheduling problems with optional tasks.[1] Clearly the modeler could directly use MiniZinc rather than MiniZinc$^+$ to model their problem (since MiniZinc$^+$ is implemented by translation) but the direct model is much less concise and much harder to get right since extended types can interact in complex ways. Our experience in creating large models using extended types by hand was that it was very difficult, motivating our need for this work.

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

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

The main contributions of the project have been presented in the workshop "ModRef 2013: The Twelfth International Workshop on Constraint Modelling and Reformulation"[20] and as a conference in "PPDP 2013: 15th Inter-

[1] Although for these scheduling problems there are approaches [19] which support stronger propagation.

national Symposium on Principles and Practice of Declarative Programming ” [21] congress.

6.2 Future Work

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 modeling more complex problems. We also plan to extend our proposal to allow union types [22].

A formalization of the evaluation and transformation of functions and predicates would complete the proposal, leading to a deeper understanding on partial functions. A implementation of this transformation using Common Subexpression Elimination [2] will decrease the size of the transformed models, possibly improving their solving performance with respect to current implementation.

Another possible application of our project is the representation of partial functions as total functions with an extra “undefined” value. This would possibly allow the modeler to decide among the different proposed semantics for undefinedness in Constraint Programing Languages [11, 10].

Bibliography

[1] 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*, pp. 529–543, Springer, 2007.

[2] 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, p. to appear, Springer, 2013.

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

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

[5] M. Fisch and A. Turquette, "Peirce's triadic logic," *Transactions of the Charles S. Peirce Society*, vol. 2, no. 2, pp. 71–85, 1966.

[6] F. Azevedo, "Thesis: Constraint solving over multi-valued logics - application to digital circuits," *AI Commun.*, vol. 16, no. 2, pp. 125–127, 2003.

[7] M. Baaz and T. U. Wien, "Multlog 1.0: Towards an expert system for many-valued logics," 1996.

[8] F. F. Liu and D. H. Moore, "An implementation of kripke-kleene semantics," *Information Sciences*, vol. 108, no. 1-4, pp. 31 – 50, 1998.

[9] E. F. Codd, "Missing information (applicable and inapplicable) in relational databases," *SIGMOD Record*, vol. 15, no. 4, pp. 53–78, 1986.

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

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

[12] Kim Marriott and Peter J. Stuckey, *MiniZinc Tutorial*. MiniZinc, 2013.

[13] T. F. Project, "flex: The Fast Lexical Analyzer."

[14] Various Contributors and the GNU Project, *Bison - GNU parser generator*. Free Software Foundation, Inc., 1998-2012.

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

[16] E. F. Codd, "Extending the database relational model to capture more meaning," *ACM Trans. Database Syst.*, vol. 4, pp. 397–434, Dec. 1979.

[17] C. Schulte, M. Z. Lagerkvist, and G. Tack, "Gecode." http://www.gecode.org/.

[18] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," *Journal of the Assotiation for Computing Machinery*, vol. 40, no. 1, pp. 143–184, 1993.

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

[20] R. Caballero, P. J. Stuckey, and A. Tenorio-Fornés, "Finite Type Extensions in Constraint Programming," in *ModRef 2013: The Twelfth International Workshop on Constraint Modelling and Reformulation*, 2013. to appear.

[21] R. Caballero, P. J. Stuckey, and A. Tenorio-Fornés, "Finite Type Extensions in Constraint Programming," in *Proceedings of the 15th International Symposium on Principles and Practice of Declarative Programming (PPDP 2013)*, ACM International Conference Proceeding Series, ACM, 2013. to appear.

[22] R. Harper, *Practical Foundations for Programming Languages*, ch. 12. Cambridge University Press, 2012.

Appendix A

MiniZinc^{+} code of Model "Board"

```
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"];
```

Appendix B

Model "SQL_or" transformation

```
var 0..1:T_a_0e0;
var 1..2:T_a_0s0;
var 0..1:T_b_0e0;
var 1..2:T_b_0s0;
var 0..1:T_c_0e0;
var 1..2:T_c_0s0;
function var bool: funs1 (var 1..2:xs0 ,var 0..1:xe0 ,var
    1..2:ys0 ,var 0..1:ye0 ) =
let { var bool:c1Let0,var bool:rLet0s0,var 0..1:rLet0e0,
    constraint '->'('!='(rLet0e0,0),'='(rLet0s0,false))} in (
    let { constraint '/\'('='(c1Let0,'/\'('='(xe0,0),'='(ye0
    ,0))),'='(0,0)),constraint '\/'('/\'(not(c1Let0),'/\'('='(
    rLet0s0,false),'='(rLet0e0,1))),'/\'(c1Let0,'='(rLet0s0,
    not('='(xs0,ys0))))))} in (rLet0s0));
function var 0..1: fune1 (var 1..2:xs0 ,var 0..1:xe0 ,var
    1..2:ys0 ,var 0..1:ye0 ) =
let { var bool:c1Let0,var bool:rLet0s0,var 0..1:rLet0e0,
    constraint '->'('!='(rLet0e0,0),'='(rLet0s0,false))} in (
    let { constraint '/\'('='(c1Let0,'/\'('='(xe0,0),'='(ye0
    ,0))),'='(0,0)),constraint '\/'('/\'(not(c1Let0),'/\'('='(
    rLet0s0,false),'='(rLet0e0,1))),'/\'(c1Let0,'='(rLet0s0,
    not('='(xs0,ys0))))))} in (rLet0e0));
function var bool: funs2 (var bool:a1s0 ,var 0..1:a1e0 ,var
    bool:b1s0 ,var 0..1:b1e0 ) =
let { var bool:c11Let0,var bool:c21Let0,var bool:r1Let0s0,var
    0..1:r1Let0e0,constraint '->'('!='(r1Let0e0,0),'='(
    r1Let0s0,false))} in (let { constraint '='(c11Let0
    ,'/\'('='(a1e0,0),'='(b1e0,0))),constraint '='(c21Let0
    ,'\/'('/\'('='(a1s0,fals),'='(a1e0,0)),'/\'('='(b1s0,fals)
```

```
    ,'='(b1e0,0)))),constraint '\/'('/\'(c11Let0,'/\'('='(
    r1Let0s0,'/\'(a1s0,b1s0)),'='(r1Let0e0,0))),'\/'('/\'(not(
    c11Let0),'/\'(c21Let0,'/\'('='(r1Let0s0,fals),'='(r1Let0e0
    ,0)))),'/\'(not(c11Let0),'/\'(not(c21Let0),'/\'('='(
    r1Let0s0,false),'='(r1Let0e0,1))))))} in (r1Let0s0));
function var 0..1: fune2  (var bool:a1s0 ,var 0..1:a1e0 ,var
    bool:b1s0 ,var 0..1:b1e0 ) =
let { var bool:c11Let0,var bool:c21Let0,var bool:r1Let0s0,var
    0..1:r1Let0e0,constraint '->'('!='(r1Let0e0,0),'='(
    r1Let0s0,false))} in (let { constraint '='(c11Let0
    ,'/\'('='(a1e0,0),'='(b1e0,0))),constraint '='(c21Let0
    ,'\/'('/\'('='(a1s0,fals),'='(a1e0,0)),'/\'('='(b1s0,fals)
    ,'='(b1e0,0)))),constraint '\/'('/\'(c11Let0,'/\'('='(
    r1Let0s0,'/\'(a1s0,b1s0)),'='(r1Let0e0,0))),'\/'('/\'(not(
    c11Let0),'/\'(c21Let0,'/\'('='(r1Let0s0,fals),'='(r1Let0e0
    ,0)))),'/\'(not(c11Let0),'/\'(not(c21Let0),'/\'('='(
    r1Let0s0,false),'='(r1Let0e0,1))))))} in (r1Let0e0));
function var bool: funs0  (var 1..2:xs0 ,var 0..1:xe0 ,var
    1..2:ys0 ,var 0..1:ye0 ) =
let { var bool:c1Let0,var bool:rLet0s0,var 0..1:rLet0e0,
    constraint '->'('!='(rLet0e0,0),'='(rLet0s0,false))} in (
    let { constraint '/\'('='(c1Let0,'/\'('='(xe0,0),'='(ye0
    ,0))),'='(0,0)),constraint '\/'('/\'(not(c1Let0),'/\'('='(
    rLet0s0,false),'='(rLet0e0,1))),'/\'(c1Let0,'/\'('='(
    rLet0s0,'/\'('='(xs0,ys0),'='(xe0,ye0))),'='(rLet0e0,0))))
    } in (rLet0s0));
function var 0..1: fune0  (var 1..2:xs0 ,var 0..1:xe0 ,var
    1..2:ys0 ,var 0..1:ye0 ) =
let { var bool:c1Let0,var bool:rLet0s0,var 0..1:rLet0e0,
    constraint '->'('!='(rLet0e0,0),'='(rLet0s0,false))} in (
    let { constraint '/\'('='(c1Let0,'/\'('='(xe0,0),'='(ye0
    ,0))),'='(0,0)),constraint '\/'('/\'(not(c1Let0),'/\'('='(
    rLet0s0,false),'='(rLet0e0,1))),'/\'(c1Let0,'/\'('='(
    rLet0s0,'/\'('='(xs0,ys0),'='(xe0,ye0))),'='(rLet0e0,0))))
    } in (rLet0e0));
function var bool: funs3  (var bool:aas0 ,var 0..1:aae0 ,var
    bool:bbs0 ,var 0..1:bbe0 ) =
let { var bool:cc1Let0,var bool:cc2Let0,var bool:rrLet0s0,var
    0..1:rrLet0e0,constraint '->'('!='(rrLet0e0,0),'='(
    rrLet0s0,false))} in (let { constraint '='(cc1Let0
    ,'/\'('='(aae0,0),'='(bbe0,0))),constraint '='(cc2Let0
    ,'\/'('/\'('='(aas0,tru),'='(aae0,0)),'/\'('='(bbs0,tru)
    ,'='(bbe0,0)))),constraint '\/'('/\'(cc1Let0,'/\'('='(
    rrLet0s0,'\/'('/\'('='(aas0,tru),'='(aae0,0)),'/\'('='(
    bbs0,tru),'='(bbe0,0)))),'='(rrLet0e0,0))),'\/'('/\'(not(
    cc1Let0),'/\'(cc2Let0,'/\'('='(rrLet0s0,tru),'='(rrLet0e0
    ,0)))),'/\'(not(cc1Let0),'/\'(not(cc2Let0),'/\'('='(
    rrLet0s0,false),'='(rrLet0e0,1))))))} in (rrLet0s0));
function var 0..1: fune3  (var bool:aas0 ,var 0..1:aae0 ,var
    bool:bbs0 ,var 0..1:bbe0 ) =
```

```
let { var bool:cc1Let0,var bool:cc2Let0,var bool:rrLet0s0,var
    0..1:rrLet0e0,constraint '->'('!='(rrLet0e0,0),'='(
    rrLet0s0,false))} in (let { constraint '='(cc1Let0
    ,'/\'('='(aae0,0),'='(bbe0,0))),constraint '='(cc2Let0
    ,'\/'('/\'('='(aas0,tru),'='(aae0,0)),'/\'('='(bbs0,tru)
    ,'='(bbe0,0)))),constraint '\/'('/\'(cc1Let0,'/\'('='(
    rrLet0s0,'\/'('/\'('='(aas0,tru),'='(aae0,0)),'/\'('='(
    bbs0,tru),'='(bbe0,0)))),'='(rrLet0e0,0))),'\/'('/\'(not(
    cc1Let0),'/\'(cc2Let0,'/\'('='(rrLet0s0,tru),'='(rrLet0e0
    ,0)))),'/\'(not(cc1Let0),'/\'(not(cc2Let0),'/\'('='(
    rrLet0s0,false),'='(rrLet0e0,1)))))))} in (rrLet0e0));
constraint '->'('!='(T_c_0e0,0),'='(T_c_0s0,1));
constraint '->'('!='(T_b_0e0,0),'='(T_b_0s0,1));
constraint '->'('!='(T_a_0e0,0),'='(T_a_0s0,1));
constraint funs2(true,0,funs3(funs1(T_a_0s0,T_a_0e0,T_b_0s0,
    T_b_0e0),fune1(T_a_0s0,T_a_0e0,T_b_0s0,T_b_0e0),funs1(
    T_a_0s0,T_a_0e0,T_c_0s0,T_c_0e0),fune1(T_a_0s0,T_a_0e0,
    T_c_0s0,T_c_0e0)),fune3(funs1(T_a_0s0,T_a_0e0,T_b_0s0,
    T_b_0e0),fune1(T_a_0s0,T_a_0e0,T_b_0s0,T_b_0e0),funs1(
    T_a_0s0,T_a_0e0,T_c_0s0,T_c_0e0),fune1(T_a_0s0,T_a_0e0,
    T_c_0s0,T_c_0e0)));
output([" INSERT INTO T (a,b,c) VALUES (", if fix(T_a_0e0 ==
    0) then show(T_a_0s0) else "ext("++ show(T_a_0e0)++")"
    endif, ",", if fix(T_b_0e0 == 0) then show(T_b_0s0) else "
    ext("++ show(T_b_0e0)++")" endif, ",", if fix(T_c_0e0 ==
    0) then show(T_c_0s0) else "ext("++ show(T_c_0e0)++")"
    endif, ");\n"]);
solve satisfy;
```