Unit Types for MiniZinc

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Jason Nguyen
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Peter J. Stuckey
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Guido Tack
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Abstract

Discrete optimization models often reason about discrete sets of objects, but discrete optimization solvers only deal with integers. One of the key challenges when building models for discrete optimization problems is avoiding bugs. Because the model only defines constraints, decisions, and an objective that are then run on a solver, bugs in the model can be very difficult to track down. Hence, modelling languages should have strong type systems to detect as many bugs as possible at the modelling level. In this paper, we propose unit types for MiniZinc. Unit types allow us to differentiate between different integers appearing in the model. Almost all integer decisions in models are either about a set of objects or some measurable resource type. Using unit types, we can add more type safety to our models by avoiding confusion of decisions on different resource types. Compared to other programming languages, unit types in our proposal are unusual. MiniZinc models often deal with multiple levels of granularity of the same resource, e.g., scheduling to the minute, but doing resource allocation on the half day, or use an unspecified granularity, e.g., the same job-shop scheduling model could use task durations given in minutes or days. Our proposed unit types also differentiate between *coordinate* unit types, e.g., the time when an event occurred, and the usual *delta* unit types, e.g., the time difference between two events. Errors arising from mixing coordinate and delta types can be very challenging to debug, so we extend the type system to track this for us.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages

Keywords and phrases Modelling, Type Safety, Unit Types

Digital Object Identifier 10.4230/LIPIcs.CP.2025.10

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

Consider the following MiniZinc model of the knapsack problem where we need to choose exactly `k` different products from a given set `PRODUCT`, so that the total weight is under the limit, and we maximize profit.

```
1 int: k;        % number of products to choose
2 int: limit;    % available weight limit
3 enum PRODUCT; % set of products available
4 array[PRODUCT] of int: wght;
5 array[PRODUCT] of int: profit;
6 array[1..k] of var PRODUCT: chosen;
7 constraint all_different(chosen);
8 constraint sum(i in 1..k)(profit[chosen[i]]) <= limit;
9 solve maximize sum(i in 1..k)(profit[chosen[i]]);
```

The model runs and will give us answers, but unfortunately it does not define the correct problem. The modeller has used `profit` instead of `wght` on line 8. In line 8 we are summing up profits to see that they are under the weight limit. This is a *unit type error* of the model, comparing integers that represent two different kinds of things.

While there are unit type extensions for almost any programming language, in practice they are not commonly used. The problem is that the *overhead* of adding unit types to the program has to pay off in comparison to the extra development time to be worth it. For safety critical code this is clearly worthwhile (as the sad demise of the $125M Mars Climate Orbiter indicates[1]). Given that debugging constraint models can be quite difficult, particularly if the solver simply fails after a large amount of computation, an important role of types in modelling languages is to provide type safety. Many subtle errors can be avoided if we use strong type checking based on the types. Here we introduce unit types for MiniZinc. Interestingly, the question of unit types for *modelling languages* for combinatorial problems is different from that of procedural languages, and we find the necessity for introducing new kinds of unit types to help prevent type errors in models.

The contributions of this paper are:

- A proposed definition of unit types for MiniZinc, including dimensions and units.
- The introduction of *abstract units*, and coercions between them, which results from the fact that we are often dealing with integer decisions, so arbitrary multiplication or division of units is not obviously meaningful.
- The introduction of *counting types*, used in models which count the occurrence of objects, a new form of unit type.
- The introduction of *coordinate* instead of the usual *difference* unit types to differentiate variables representing coordinates from those representing distances. These have been defined for other programming languages but less frequently.
- The definition of a unit type checker based on type erasure at the model level, resulting in zero overhead at solving time compared to models that do not use unit types.
- An exploration of existing MiniZinc models to determine how many of them could improve type safety utilizing unit types.

2 Unit Types

Unit types are refinements of numeric types that attach units of measurement to the numeric objects. They aim to find more type errors. Note that we introduce unit types as an addition on top of the current MiniZinc type system. Although the enhanced type system might reject models that use unit types, it does not influence the correctness of existing MiniZinc models.

[1] An investigation indicated that the failure resulted from a navigational error due to commands from Earth being sent in English units (in this case, pound-seconds) without being converted into the metric standard (Newton-seconds).

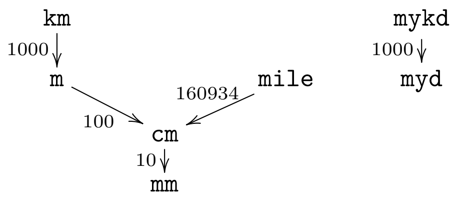

Figure 1 Graph for dimension `distance`.

A *dimension* is a kind of measurement, for example distance, time, mass, and worth are basic dimensions. Dimensions can be declared as "`unit type` *dimension*". We assume m distinct dimensions are available. A *basic unit* is an identifier such as `metre` (metre), `sec` (second), `kg` (kilogram), or `dollar` which has a dimension. A basic unit type can then be *declared* as "`unit` *dimension*`:` u", where $dim(u) = dimension$. Or it can also be *derived* from an earlier declared or defined unit type u' as "`unit` *dimension*`:` u `=` c`@`u'" where $dim(u) = dim(u') = dimension$. The basic unit types for a dimension *dimension* define a graph with a node for each basic unit type and edges (u, u') with weight c for each declared unit type definition.[2]

One basic unit type u can be *downcast* to another v of the same dimension by multiplying by the product of constants appearing in the path from u to v, denoted $\downarrow(u, v)$, which equals $\perp$ if no path exists. We use $\perp$ to represent a type error/ill-defined type. We define the *meet*, largest common subunit, of two basic unit types of the same dimension $\sqcap_b(u, v)$ as the nearest common ancestor of u and v in the graph, or $\perp$ if none exists.

Figure 1 shows the graph resulting for dimension `distance` from the sample declarations on the left below. For example, $\sqcap_b($`km`,`mile`$) =$ `cm`, and $\sqcap_b($`km`,`mykd`$) = \perp$; similarly $\downarrow($`mile`,`mm`$)$ $= 1609394 \times 10$, while $\downarrow($`mm`,`km`$) = \perp$. Note how we can have multiple declared basic units for the same dimension, but we cannot coerce from one to the other.

```
unit type distance;                unit type time;
unit distance: mm;                 unit time: sec;
unit distance: cm = 10@mm;         unit time: minute = 60@sec;
unit distance: metre = 100@cm;     unit time: hour = 60@minute;
unit distance: km = 1000@m;        unit type: mass;
unit distance: mile = 160934@cm;   unit mass: gram;
unit distance: myd;                unit mass: kg = 1000@gram;
unit distance: mykd = 1000@myd;    unit type worth;
                                   unit worth: dollar;
```

A (complex) unit type u for a numeric object o takes the form $u \equiv b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$, where each $b_i, 1 \leq i \leq m$ is a basic unit type for dimension i, and each $n_i, 1 \leq i \leq m$ is an integer. The dimension of the type u is $dim(b_1)^{n_1} \cdots dim(b_2)^{n_m}$. For example, a speed may have unit type `metre/sec`, which is technically $\texttt{metre}^1\texttt{sec}^{-1}\texttt{kg}^0\texttt{dollar}^0$ and has dimension $\texttt{distance}^1\texttt{time}^{-1}\texttt{mass}^0\texttt{worth}^0$. Note that we do not allow complex unit types with two basic unit types of the same dimension. We usually omit writing terms where $n_i = 0$. We denote by $\mathbf{1}$ the dimensionless unit type (where $n_i = 0, 1 \leq i \leq m$).

[2] Usually c is an integer. Floating-point constants c are allowed, but they result in automatic casts of integer expressions to float.

We can add shorthands for dimension expressions and (complex) unit type expressions. A declaration "`unit type` *short* `=` *de*" defines *short* as a shorthand for the dimension expression *de*. *de* takes the form of *de*`*`*de*, *de*`/`*de*, de^n, or d where d is a previously defined dimension or shorthand and n is an integer. Similarly, (complex) unit type shorthands can be defined as "`unit` *dimension*`:` u `=` *ue*" where *ue* is defined as above except that d must be a basic or shorthand unit type, and $dim(ue) = dimension$. For example, the declarations below define velocity as a shorthand for distance over time, and a unit of velocity which is always treated as its two components.

```
unit type velocity = distance/time;
unit velocity: kph = km/hour;
```

We extend the meet operation to a single dimension with power as follows: $\sqcap_d(b_i^{n_i}, {b'_i}^{n'_i})$ equals $\perp$ if $dim(b_i) \neq dim(b'_i)$ or $n_i \neq n'_i$; otherwise it equals $\sqcap_b(b_i, b'_i)^{n_i}, n_i \geq 0$ or $b_i^{n_i}$ if $\downarrow(b_i, b'_i) \neq \perp$, or ${b'_i}^{n_i}$ if $\downarrow(b'_i, b_i) \neq \perp$, or $\perp$ otherwise. So for positive powers we use the meet operation, for negative powers we require that one unit is downcastable to the other, and the resulting type is the downcastable type, inverse powers swap the direction to smaller units.

We can then extend meet to two complex unit types: $\sqcap(b_1^{n_1} \cdots b_m^{n_m}, c_1^{k_1} \cdots c_m^{k_m})$ is defined as $\Pi_i \sqcap_d (b_i^{n_i}, c_i^{k_i})$ if none of the dimensional meets results in $\perp$, and $\perp$ otherwise. We can similarly extend downcasting from one complex type to another as $\downarrow(b_1^{n_1} \cdots b_m^{n_m}, c_1^{k_1} \cdots c_m^{k_m})$ as $\Pi_{i,n_i>0} \downarrow(b_i, c_i)^{n_i} \times \Pi_{i,n_i<0} \downarrow(c_i, b_i)^{-n_i}$ if no downcasts results in $\perp$ and $n_i = k_i, 1 \leq i \leq m$, or $\perp$ otherwise. This allows us to, for example, determine that $\sqcap(\texttt{km}^2\texttt{sec}^{-1}, \texttt{m}^2\texttt{hour}^{-1})$ is $\texttt{m}^2\texttt{hour}^{-1}$ and $\downarrow(\texttt{km}^2\texttt{sec}^{-1}, \texttt{m}^2\texttt{hour}^{-1})$ is $1000^2 \times (60 \times 60)^{--1} = 3600000000$.

We can multiply two complex unit types $u \otimes v$ together: $(b_1^{n_1} \cdots b_m^{n_m}) \otimes (c_1^{k_1} \cdots c_m^{k_m})$. This is defined as $x_1^{n_1+k_1} \cdots x_m^{n_m+k_m}$, where $x_i = b_i$ if $b_i = c_i \vee k_i = 0$, and $x_i = c_i$ if $n_i = 0$, and the result is $\perp$ if neither of these conditions hold for any dimension. Note that this *requires* that we do not multiply unit types with different basic types for the same dimension together.[3] We can invert a unit type $u \equiv b_1^{n_1} \cdots b_m^{n_m}$ to obtain $\mathbf{1}/u$ as $b_1^{-n_1} \cdots b_m^{-n_m}$. Division of unit types is just inversion followed by multiplication: $u/v = u \otimes (\mathbf{1}/v)$. For example, $\texttt{km}^1\texttt{sec}^{-1} \otimes \texttt{km}^1$ gives unit type $\texttt{km}^2\texttt{sec}^{-1}$ and $\mathbf{1}/(\texttt{km}^1\texttt{sec}^{-1}) = \texttt{km}^{-1}\texttt{sec}^1$.

Note that while meet and down casting are a bit complex to define, actual models mainly use simple cases, as there are usually one or two basic unit types for each dimension.

3 Unit Type Checking

The main role of unit types is to prevent making unit type errors. They restrict the possible correct arithmetic expressions we can write. Below, we give our candidate syntax to work with unit types in existing MiniZinc models, and the type computation rules for important classes of arithmetic expression. Importantly, unit types are checked at the model level (without the data) and then erased, incurring no runtime overhead when solving the problem.

A constant c can be created of arbitrary unit type u using the `@` symbol, c`@`u. The operator should have a high precedence to ensure it binds tightly to prevent confusion about which expression it influences. A variable can be declared with a unit type by appending the unit type onto the type declaration using the `@` symbol as follows: "[`var`][`int`|`float`]`@`u`:`*id*"

[3] It is possible to introduce automatic coercion during multiplication, but there are quite a few non-obvious cases, so at present we restrict multiplication. The coercions could always be explicitly added to the model to allow a multiplication.

declares identifier *id* to have unit type u. We can similarly define set or array arithmetic parameters or decisions with unit types. During type inference/checking, the type of identifier *id* is its declared unit type u.

Here are the most important typing rules for this new syntax. The complete set of typing rules is given in Appendix B.

- $\mathit{type}(k@u) = u$. Constants simply have the annotated unit type.
- $\mathit{type}(e_1 + e_2) = \sqcap(\mathit{type}(e_1), \mathit{type}(e_2))$. Addition (and subtraction) require unit types which have a meet (i.e. they have the same dimensions and there is a coercion to a common unit).
- $\mathit{type}(e_1 \times e_2) = \mathit{type}(e_1) \otimes \mathit{type}(e_2)$. Multiplication of unit types is free in the dimension of the indices but restrictive in the basic unit types.
- $\mathit{type}(e_1 \texttt{ div } e_2) = \mathit{type}(e_1) \otimes \mathit{type}(\mathbf{1}/e_2)$. Division of unit types is simply inversion followed by multiplication.

Most other integer operations act similarly to $+$. For example, max, min and `if-then-else-endif` have type given by the meet of their (last 2) arguments.

Returning to the example from the introduction, we might add units types as follows

```
include "units.mzn";   % make declared units available
int@kg: limit;         % available weight limit
array[PRODUCT] of int@kg: wght;
array[PRODUCT] of int@dollar: profit;
```

The first line makes the standard library unit declarations available to the model. The parameters are now specified in terms of units. With these declarations, we immediately get the following unit type error on the second last line of the original model: `unit mismatch: expected "dollar", but got "kg"`.

To produce meaningful error messages for unit mismatches, the original units as written by the modeller (or propagated through multiplication or division) are always used in any user-facing messaging, in addition to the computed normalized units. This ensures that the modeller is not faced with error messages only mentioning constructs they never wrote, and the normalized units allow for easy comparison of the two units to see the exact mismatch. For example, for the following model an error will be produced for the last line stating: `unit mismatch: expected "vel = "m*s^-1" but got "vel/s = m*s^-2"`.

```
unit velocity: vel = m / s;
float@m: a = 5@m;
float@s: b = 2@s;
float@vel: c = a / b;
float@vel: d = c / b;
```

3.1 Coercion

In usual unit systems, coercion from one unit to a unit of the same dimension is generally allowed. The physical quantities being measured by dimensions are typically real, and are represented by a form of floating point number. This means that creating large or small values is not problematic. Even in the case that the dimension is arguably not a real number (e.g., worth might only be meaningful down to 1¢), it is still generally acceptable to use its smallest unit type (using large integers to represent it).

Unfortunately, this is not a suitable approach for (discrete) modelling languages such as MiniZinc. Most MiniZinc models contain integer decision variables, and using the smallest unit type to represent them can immensely expand the search space. This, in turn, would lead to large increases in solving time. Hence, we need to be conservative with automatic casting between units.

We propose a system that only allows automatic *downcasts* of units over the same dimension from a larger unit to a smaller unit, e.g. from `kg` to `gram`.

```
var int@kg: x;
var int@gram: y = x + 55@gram;
var int@kg: z = x + y;
```

which will in effect constrain $y = 1000 \times x + 55$, while the last line with create a unit type error, since $x + y$ has unit type `gram` and we cannot automatically upcast to fit into `kg`.

The user can instead *upcast* from a smaller unit to a larger unit when required using provided built-in upcast operations: `round`, `ceil` and `float`. We can correct the error above by rewriting it as

```
var int@kg: z = ceil(kg, x + y);
```

which will constrain $z = \lceil\frac{1000\times x+y}{1000}\rceil$. The compiler can automatically create the underlying upcast function `ceil_kg_g` and replace the call in the original code by looking up the appropriate conversion factor:

```
var int@kg: ceil_kg_g(var int@gram: a) = ceil(a / 1000@g) *1@kg;
```

Automatic downcasts should be added every time a meet operation does not result in the same type as its arguments. So an expression $x + y$ where $type(x) = u$ and $type(y) = v$ and $\sqcap(u, v) = w$ will be replaced by $\downarrow(u, w)*x + \downarrow(v, w)*y$. For example, assuming $type(p) =$ `mile`, the expression `2@km + p@mile`, will be rewritten to $1000 \times 100 \times 2 + 160934 \times p$ with unit type `cm`. Note that if all constants in derived unit type declarations are integers, then automatic downcasts of an integer expression always remain integer.

Consider a model which schedules tasks to the minute, but tracks machine usage over each half day in the planning period (where tasks cannot span across multiple half days). A model that computes in which half days a machine is used from the start times of the tasks is given below.

```
unit time: halfday = 12@hour;
int@minute: makespan;
set of int@minute: TIME = 1@minute..makespan;
int@halfday: halfdays = ceil(halfday,makespan);
set of int@halfday: HALF = 1@halfday..halfdays;
array[TASK] of var TIME: start;
array[TASK] of var MACHINE: machine;
array[MACHINE,HALF] of var bool: used;
constraint forall(t in TASK)
                    (used[machine[t], ceil(halfday, start[t])]);
```

The translation from start times in minutes to the correct halfday is easily defined by the user using `ceil` upcasting. Note that we cannot know which rounding semantics is correct for this upcast automatically, so we must rely on the modeller.

Once unit type checking is complete, we add the multiplications required by an automatic downcasts that arose in the unit type checking, and add definitions for any upcast predicates used by the modeller. This is the only stage where code needs to be generated. After this, all unit types can be *erased* from the model allowing all following compiler phases to continue unaffected. For example, the model

```
var int@kg: x;
var int@gram: y = x + 55@gram;
var int@kg: z = ceil(kg,x + y);
```

will produce the following after unit coercion and erasure.

```
var int: x;
var int: y = x * 1000 + 55;
var int: z = ceil_kg_g(100*x + y);
function var int: ceil_kg_g(var int: a) = ceil(a / 1000);
```

3.2 Abstract units

Most MiniZinc models do not reason about a dimension using two different units, but we often reason about a dimension without specifying exactly what it means. For example, the knapsack problem of the introduction is equally valid whatever unit we describe mass in, as long as we use the same unit for both `limit` and `wght`. Similarly, the unit of worth could be cents or millions of dollars without changing how the model works. Hence, models can define *abstract* units. This does not require any extension to the language. Instead, we simply define new basic unit types in the model itself. For our knapsack example, we could introduce a new unit for mass `mmass` and worth `mworth` as follows.

```
1 include "units.mzn";  % make declared dimensions available
2 unit mass: mmass;
3 unit worth: mworth;
4 int@mmass: limit;     % available weight limit
5 array[PRODUCT] of int@mmass: wght;
6 array[PRODUCT] of int@mworth: profit;
```

Every dimension automatically creates a *default* abstract unit of the same name. Since in most models we only reason about one unit for a dimension, we can just use this unit. For example in the above example, we could remove lines 2 and 3, and replace `mmass` by `mass` and `mworth` by `worth` to use the default abstract units.

3.3 Units and Data in MiniZinc

Since data for discrete optimization models is usually created outside the model in existing unitless formats like CSV, JSON, or a database, it is important that we support unitless data. We assume that datafiles are unitless, and the unit information is attached to the definition of parameters. For our first example model, a sample (`.dzn`) data file might be:

```
k = 3;
PRODUCT = {cooktop, stove, oven, fridge, washer, dryer};
limit = 170;
wght = [16, 70, 32, 90, 70, 30];
profit = [400, 750, 500, 1200, 600, 500];
```

In the future, interfaces that use MiniZinc from other languages with unit types, such as MiniZinc Python [6], could provide information about the unit types of the data.

4 Counting types

One of the differences of discrete optimization models from normal programs is that they often reason about counts of different kinds of objects. We want to ensure that we do not make unit errors in our model when dealing with counting. Consider a variation of our running model where we use 0/1 variables to define the chosen products.

```
1 int: k;        % number of products to choose
2 int: limit;    % available weight limit
3 enum PRODUCT; % set of products available
4 array[PRODUCT] of int: wght;
5 array[PRODUCT] of int: profit;
6 array[PRODUCT] of var 0..1: chosen;
7 constraint sum(chosen) = k;
8 constraint sum(p in PRODUCT)(chosen[p]*wght[p]) <= limit;
9 solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

What unit type do `k` and the `chosen` variables have? They are *counts* of `PRODUCT`. We extend any enumerated type in MiniZinc to also be used as the *unit type* for counting that enumerated type. In this model, the suggested unit type declarations are

```
int@PRODUCT: k;      % number of products to choose
int@mmass: limit;    % available weight limit
array[PRODUCT] of int@(mmass/PRODUCT): wght;
array[PRODUCT] of int@(mworth/PRODUCT): profit;
array[PRODUCT] of var (0..1)@PRODUCT: chosen;
```

The `wght` array records *mass per PRODUCT*, while the `profit` array records *worth per PRODUCT*. Since we are multiplying the counts of `PRODUCT`s `chosen[p]` by the product's weight `wght[p]`, it is clear that the constraint on line 8 is unit correct.

Note that the two declarations `PRODUCT: k` and `int@PRODUCT: k` are quite different, the first `k` holds a (non-numeric) product, while the second `k` holds an integer *count* of products.

A counting type is simply a new dimension together with a basic unit. So each enumerated type `E` adds a new basic unit also named `E` and a new dimension also named `E` to the unit type system, where $dim(\mathtt{E}) = \mathtt{E}$.

4.1 Fine counting types

We can actually still generate erroneous models even with the counting types we consider above. Consider the following model, which is checking that for any two chosen products their usage of any resource is below the pairwise usage limit for that resource:

```
enum RESOURCE;
array[PRODUCT, RESOURCE] of int@(RESOURCE/PRODUCT): usage;
array[RESOURCE] of int@RESOURCE: limit2;
array[PRODUCT] of var (0..infinity)@PRODUCT: chosen;
constraint forall(p1, p2 in PRODUCT where p1 < p2,
                  r in RESOURCE)
                 (usage[p1,r]*chosen[p1] +
                  usage[p2,r]*chosen[p1] <= limit2[r]);
```

This is unit type correct, but unfortunately, there is a copy and paste bug. The problem here is that the counting types are too broad. The erroneous expression `usage[p2,r] * chosen[p1]` talks about different products. These types of errors are easy to make when we reason about two or more resource types in the same arithmetic expression.

To prevent these errors, we allow fine counting types for each member of an enum, as long as we have the enum as an index in the expression. We can rewrite the model declarations

```
enum RESOURCE;
array[p of PRODUCT, rr of RESOURCE] of int@(rr/p): usage;
array[rr of RESOURCE] of int@rr: limit2;
array[p of PRODUCT] of var (0..1)@p: chosen;
```

The notation v `of` E gives us access to the value v of the enumerated type E in the rest of the declaration.[4] In this declaration, we are explicit that each value in the `usage` array is specific to the PRODUCT and RESOURCE combination. It defines the resource usage per product. We similarly define the resource limits and the chosen resources. With these unit type definitions we can detect that the unit type of `usage[p1,r]*chosen[p1]` is `r`, but the unit type of `usage[p2,r]*chosen[p1]` is `r/p2*p1`, and therefore, the last constraint is erroneous. The right-hand side of the `+` operator is giving a unit mismatch error: `expected "r" but got "r*p2^-1*p1"`.

In the formalization, each fine-grained unit type used in the model, e.g., `p of PRODUCT`, defines a new basic unit type $PRODUCT(e)$ of dimension $PRODUCT$ for each expression e appearing in an array lookup at that position. For the example above, we have new unit types $PRODUCT(p1)$, $PRODUCT(p2)$ and $RESOURCE(r)$. The type of `usage[p2,r]` is obtained by replacing the p by $p2$ and r by rr giving $PRODUCT(p2)/RESOURCE(r)$, similarly the type of `chosen[p1]` is $PRODUCT(p1)$, and the result of the multiplication is invalid (two units for the same dimension). The printed error message omits the dimension annotation around the units.

There is no coercion from fine-grained unit type `p` to the coarse grained unit type `PRODUCT`, since allowing it would remove the unit type error in the example above, by coercing both unit type `p1` and unit type `p2` to `PRODUCT`.

5 Coordinate types

Consider the following MiniZinc model for a simple scheduling problem with precedences between pairs of tasks:

```
enum TASK;                           % set of tasks
array[TASK] of int@minute: d;        % duration of task (mins)
enum PREC;                           % set of precedences
array[PREC, 1..2] of TASK: pt;       % set of task pairs
array[TASK] of var int@minute: s;    % start time decisions
constraint forall(p in PREC)
                   (s[pt[p,1]] + d[pt[p,1]] <= s[pt[p,2]]);
constraint disjunctive(d,s);
```

[4] Similar notation has been used for other purposes in the modelling language OPL.

Each task has a duration `d`, and we wish to decide its start time `s` (all in minutes). The constraint in lines 6-7 ensures that the first task in each row of `pt` must be completed before the second task. Finally, no two tasks should overlap. The model is unit type correct, and runs and can give answers, but it is wrong!

In MiniZinc and almost all languages, we consider the numeric types to be essentially *differences*. That is, they are a value that can be added to each other, or subtracted or multiplied. But there is a different use of numeric types as *absolute coordinates* in some numerical system. It does not make sense to add these coordinates usually: consider 11°C and 25°C, while it certainly makes sense to determine their difference $25°\mathrm{C} - 11°\mathrm{C} = 8°\mathrm{C}$, how are we to interpret $11°\mathrm{C} + 25°\mathrm{C}$? Adding two temperatures does not make any sense.

This arises in models where we are reasoning both about absolute coordinates in some unit and the differences. Coordinate unit types and difference unit types are supported in some other unit type systems [9].

Suppose we are computing the end position we reach, given a fixed duration of running at a fixed velocity, given an unknown start position. Then the start and end position are both coordinates (in some coordinate space), while the length, velocity, and duration are non-coordinate (difference) values. We use the notation `coord(`u`)` to define a coordinate unit type of basic unit type u. The model that separates these different types is written:

```
var int@coord(m): start;
var int@coord(m): end;
var int@m: len = end - start;
var int@s: duration;
int@(m/s): velocity;
constraint len = duration * velocity;
```

Coordinate types are much more restricted than ordinary unit types, they support the following typed arithmetic operations only:

- $\texttt{coord}(x) + x = \texttt{coord}(x)$
- $\texttt{coord}(x) - x = \texttt{coord}(x)$
- $\texttt{coord}(x) - \texttt{coord}(x) = x$

So we can only add or subtract a (difference) unit value to a coordinate unit value to get another coordinate, or subtract one coordinate from another to get a (difference) unit value. Importantly, we cannot add two coordinate type values together, even if they have the same unit type. Similarly, we cannot multiply or divide a coordinate type, since this is equivalent to a repeated addition or its inverse.

Coordinate types appear frequently in scheduling and packing problems within the combinatorial problem class. We can restrict our definitions of scheduling and packing global constraints to enforce the correct use of coordinate types. For example, for `disjunctive` the start times are coordinates while the durations are usual (difference) unit types. With the unit typed definition (shown in the next section) we discover the error in the model at the beginning of the section. We have reversed the arguments to disjunctive. They both have the same units, so there is no unit error, but one is a coordinate type, the other not. The corrected model rewrites these two lines:

```
array[TASK] of var int@coord(minute): s;
constraint disjunctive(s, d);
```

If we only rewrite just one of the two lines, we still get a type error.

Since coordinate types only seem to make sense for a single dimension, we restrict *coord*(u) to only be defined for declared or derived basic unit types u. Again, this restriction is made to get stricter unit types, and hence discover more errors.

6 Global constraints and Units

One of the strengths of constraint programming modelling languages including MiniZinc is the wide range of global constraints that they support. In order for units to be truly useful, we need to extend these global constraint definitions to use units accurately. To do so, we need to be able to define functions which are parametric in their unit types. We introduce (unit) type variables `$v` to appear as unit expressions. In MiniZinc the notation `$$V` indicates an arbitrary enumerated type.

6.1 Function parameters with units

There are many arithmetic constraints to which we can attach units to the arguments of interest. For example, `sliding_sum` constrains the sum of each consecutive sequence of `seq` elements in the array `vs` to be in the range `low..up`. The type of this global constraint including units is simply:

```
predicate sliding_sum(int@$u: low,
                      int@$u: up,
                      int@$$E: seq,
                      array [$$E] of var int@$u: vs)
```

Notice how `seq` is a counting type for the index set of `vs`.

Similarly, the knapsack constraint deals with at least two units: measuring worth (for the objective) and weight for the constraint. The unit typed declaration is:

```
predicate knapsack(array [$$I] of int@($W/$$I): w,
                   array [$$I] of int@($P/$$I): p,
                   array [$$I] of var int@$$I: x,
                   var int@$W: W,
                   var int@$P: P)
```

where we are explicit about the two units used for the constraint and objective. We can also define a version using fine counting types:

```
predicate knapsack(array [i of $$I] of int@($W/i): w,
                   array [i of $$I] of int@($P/i): p,
                   array [i of $$I] of var int@i: x,
                   var int@$W: W,
                   var int@$P: P)
```

Note we can use fine unit types when the index type is an enumerated type.

The counting global constraints have explicit unit types using the counting units. The `among` constraint counts how many occurrences of the elements in a subset appear in an array. Obviously if the array contains elements of an enumerated type, then it returns a count unit for that type. The parametric type of `among` is defined as

```
function var int@$$E: among(array [$X] of var $$E: x,
                            set of $$E: v)
```

We can similarly treat `global_cardinality` constraints. But the ability to use fine counting types also allows us to define a new version of `global_cardinality` when we count the elements of an enumerated type:

```
function array [t of $$T] of var int@t:
         global_cardinality(array [$X] of var $$T: x)
```

Finally, packing and scheduling constraints make use of coordinate unit types, and polymorphic unit type variables. The `disjunctive` constraint is defined as:

```
predicate disjunctive(
    array[$$T] of var int@coord($t): start,
    array[$$T] of var int@$t: duration);
```

For example, for `cumulative`, the start time is a coordinate time unit type, while duration is the usual (difference) version of the same time unit. The resource usage for each task and capacity uses the same unit for resource. This allows us to give, for example, the capacity in metric tons and the resource usage in kilograms, with coercion done at the call interface.

```
predicate cumulative(
    array[$$T] of var int@coord($t): start,
    array[$$T] of int@$t: duration,
    array[$$T] of int@$u: usage, int@$u: capacity);
```

Similarly, the `span` constraint, which enforces that the optional task defined by start `s0` and duration `d0` *spans*, i.e., goes from the earliest start time to the latest end time, of a set of optional tasks defined by start times `s` and durations `d`, combines coordinate and difference types with optionality:

```
predicate span(var opt int@coord($t): s0,
               var int@$t: d0,
               array [$$E] of var opt int@coord($t): s,
               array [$$E] of var int@$t: d)
```

7 Evaluation

We have developed a prototype implementation of the unit type extension to MiniZinc.[5] The implementation performs the type checking for the unit types, adds in the required coercions, and then erases the unit types from the model. Afterwards, type checking and compilation of MiniZinc continues as normal.

7.1 Analysis: MiniZinc Challenge

To evaluate the extent of the usefulness of unit types, we examine the MiniZinc Challenge problems from the last four years, 2021–2024, to see how they would be written with unit types. In our analysis, we exclude procedurally generated models and the `unison.mzn` model because their immense size and lack of documentation prevent us from rewriting these models. Models used in multiple years have only been included in the counts for their earliest occurrence. The increase in model character count was calculated by counting the number of non-whitespace, non-comment characters that were added to each model where unit types were applicable (reported as a percentage increase). Table 1 gives a summary of the results of the evaluation, and the models are available at `https://www.minizinc.org/unit-types/`.

In our analysis, we found that the majority of models can benefit from unit types (*Units applicable* column), e.g. 13 out of 18 models in 2021 could be rewritten to use units, and 56 out of 71 overall. Notably, the use of unitless numeric types is nearly eliminated when models

[5] The prototype is available at `https://www.minizinc.org/unit-types/`.

Table 1 A table showing the mean increase in model size for using unit types and usage of the different unit type features: **Chars** increase in non-whitespace, non-comment characters; **Bytes** increase in raw file size; **Count** counting types; **Fine** Fine counting types; **Coord** coordinate types; **Global** unit-specialized globals.

		Mean size increase		Benchmarks using the unit type feature			
Year	Units applicable	Chars	Bytes	Count	Fine	Coord	Global
2021	13/18	9.2%	3.2%	6	3	5	4
2022	19/20	8.0%	3.6%	8	3	3	9
2023	13/18	4.7%	1.4%	4	4	10	3
2024	11/15	6.9%	5.5%	1	0	5	3
Overall	56/71	7.1%	3.3%	19	10	23	19

are refactored to employ enumerated and unit types. Remaining integer expressions without unit types generally involve: counting of Boolean conditions, factors to convert between different units, or reasoning about the numbers themselves (e.g. in pure combinatorial problems). For example, the `mznc2017_aes_opt.mzn` model reasons about a cryptographic algorithm, where integers are used from the perspective of numerics. Using unit types in a model only results in a small increase in code size in most cases. In fact, they can even lead to smaller models since having the units alleviates the need to include these in variable names or comments as is often done.

We observed that use of the already existing `any` keyword for inferring variable type and unit significantly reduces repetitive specification of unit types when storing the results of expressions. Although the removal of the specific types might remove some “self-documentation” of variable declarations, modern code editors can often display the type that the compiler determined, and this could be extended to also include the inferred unit.

Furthermore, many models where counting types can be used can utilize fine-grained counting types. However, the use of fine-grained counting types over general counting types does not always translate into improved type safety. Most expressions only access a single counting element at a time. Due to the nature of the unit types implementation, it is easy for modellers to choose what counting type suits their implementation best.

The most extensive changes in models arise from the introduction of coordinate unit types. In these models, normal units and their corresponding difference types are stored together. This suggests that the introduction of coordinate types might offer an opportunity to detect many subtle problems in optimization models which reason about both coordinates and differences.

The benchmarks used in the MiniZinc Challenge are mostly designed for benchmarking purposes and also do not include any floating-point variables, so we did not see extensive use of the concrete units defined in the units type library, which are most useful when dealing with real-world physical quantities. In most cases, models were able to use the predefined dimensions from the units type library, and use their own or default abstract units.

7.2 Case Study: Equipment Placement

To illustrate the use of complex unit types, we show a MiniZinc model for placing equipment in a two-dimensional array, with restrictions on how close they can be placed in terms of Manhattan distance. This example is designed to show one of the most **intrusive** uses of unit types because we *choose* to use separate units for width and height as well as Manhattan distances. We can make a much simpler model if we only use one distance unit, but we can then mix up the usages of the three types of distance.

The three units are defined below using coordinate types for the x, y coordinates:

```
unit distance: dist;
unit distance: width;
unit distance: height;
int@width: W; % width of area
set of int@coord(width): COL = 1@width..W;
int@height: H; % height of area
set of int@coord(height): ROW = 1@height..H;
```

There are different kinds of equipment that can be placed, with data about it, the cost, availability, and effectiveness at various distances, defined using fine counting types:

```
enum EQUIP;                       % different equipment available
int@dist: maxradius;              % max protection radius
set of int@dist: DIST = 0@dist..maxradius;
unit worth: mworth;               % unit of cost
unit type measure;                % new dimension "measure"
unit measure: effect;             % unit of measure "effectiveness"
array[e of EQUIP] of int@(mworth/e): cost;       % cost
array[e of EQUIP] of int@e: avail;               % availability
array[e of EQUIP, DIST] of int@(effect/e): eff; % effect at dist
```

We are given a limit on the total number of equipment placed. The key decisions are where to place equipment up to some given limit on number, defined as an x and y position and the type of equipment placed t.

```
int@mworth: budget;               % budget on equipment
int@EQUIP: limit;                 % max number of equipment;
set of int@EQUIP: LEN = 1@EQUIP..limit;
array[LEN] of var opt COL: x;    % x position of unit (or absent)
array[LEN] of var opt ROW: y;    % y position of unit (or absent)
array[LEN] of var opt EQUIP: t; % which equipment at this pos
var 0@EQUIP..limit: used;        % number of equipment used
```

Option types allow us not to use all of `LEN` to hold actual equipment, only the positions up to `used` are actually used. Entries in these arrays at positions greater than `used` are forced to be absent:

```
constraint forall(i in LEN)(i > used <->
           absent(x[i]) /\ absent(y[i]) /\ absent(t[i]));
```

The most complex part of the unit types is computing Manhattan distances defined as:

```
function var int@dist: man(var opt int@coord(width): x1,
                           var opt int@coord(height): y1,
                           var opt int@coord(width): x2,
                           var opt int@coord(height): y2) =
    abs(x1 - x2) * 1@(dist/width) +
    abs(y1 - y2) * 1@(dist/height);
```

where we strip the width and height units and add back `dist`.[6] Note that the subtraction of one coordinate type from another to compute a non-coordinate typed result. The most complicated constraint adds up the effectiveness of each unit at each location, defined as

```
array[ROW,COL] of var (0..100)@effect: cover;
constraint forall(r in ROW, c in COL)
    (cover[r,c] = min(100@effect, sum(i in 1@EQUIP..used)
        (let { var int@dist: dst = man(x[i],y[i],r,c); } in
        if dst > maxradius then 0@effect
        else eff[t[i],dst] endif)));
```

Here for every row and column r,c we sum up the effectiveness of each placed equipment that covers that location by computing the distance to each placed equipment and, if within the max radius, adding its effectiveness, capped at 100.

Availability constraints are enforced by a unit typed global cardinality (lower and upper bounds are the 3rd and 4th arguments):

```
constraint global_cardinality(t, EQUIP, [0@e|e in EQUIP], avail);
```

Most constraints are only minutely affected by units, if at all. For example, the budget constraint, and the constraint forcing equipment to be no closer than 3 units of distance are written as follows:

```
constraint sum(i in 1@EQUIP..used)(cost[t[i]]) <= budget;
constraint forall(i, j in 1@EQUIP..used where i < j)
                  (man(x[i], y[i], x[j], y[j]) >= 3@dist);
```

The three separate units used for distance make this one of the more complicated models to add strong unit types too. The payoff is that we can no longer mix them up. The only real complexity arises from computing Manhattan distances, which encapsulated the type coercion of the distance types. Note that, finally, every integer in the model has a unit type (or is an enumerated type).

8 Related Work

The concept of unit types in programming languages has a long history. For many years, it has been clear that programming languages can help avoid crucial problems by incorporating unit types [10]. The first implementation of these features was in the Pascal and Ada languages [13, 7, 3]. Since then, extensions have been proposed for many types of programming languages, including functional languages [11, 12] and object-oriented languages [1].

Research in database systems has also explored the use of unit types. Works like [8] propose extending database schemas to include unit types. In particular, when dealing with international data, where the same data might be recorded with different units, the use of unit types enables robust data storage, retrieval, and manipulation across datasets.

These days, the incorporation of units directly within programming languages and database systems seems to have dwindled. However, this can be directly attributed to the "meta" capability of modern systems, allowing the functionality to be implemented as external libraries. The widespread adoption of libraries like Boost.Units [16], mp-units [15],

[6] The function takes *optional* integers since the x and y positions are optional, although it will never be called with absent values.

astropy.units [2], Pint [9], postgresql-unit [4], and many others [14] in various programming environments underscores the continued importance of unit type safety. These libraries provide extensive unit catalogues and enforce unit consistency during calculations, reducing the risk of errors in modern day scientific and engineering applications.

In the field of constraint modelling, the Pyomo constraint modelling library [5] is the only modelling tool that we are aware of that enables the use of unit types. Based on the Pint Python package, Pyomo allows users to specify units of decision variables and provides helper functions to check what the units of expressions are and to check whether they are consistent. Because Python and Pyomo do not use a strong typing system, the user is charged with ensuring the consistency of the model is checked.

In comparison to Pyomo, once unit types are added to a MiniZinc model, the checking of these types is always enforced by the compiler. In addition, the static type checking and type erasure in MiniZinc mean that using unit types incurs no significant time overhead during the translation. The additional unit type variants, counting types and coordinate types, are not available in Pyomo. Similar to Pyomo, MiniZinc's interfaces to other languages, such as MiniZinc Python, might in the future be extended to connect to existing unit type libraries, such as Pint. By extending these interfaces to handle unit information from other library, developers can verify that unit-aware data is consistent with their unit-aware MiniZinc model.

9 Conclusion

Unit types provide stronger type safety than is usual for even strongly typed languages. While unit type extensions have been defined for almost every programming language, they are still rarely used, even though they can detect rather subtle bugs, which can have catastrophic effect (as exemplified by the Mars Climate Orbiter debacle). The problem with unit types for developers is that the overhead of using them may not appear to pay off. The trade-off for modelling languages is much more attractive, since debugging subtle errors in a model is much more challenging than in procedural code. Interestingly, there are kinds of unit types that only arise in discrete optimization, so-called counting types. In this paper, we define a lightweight but fairly complete system of unit types suitable for discrete optimization modelling languages. We have fully implemented this as an extension of the MiniZinc modelling language. In practice, we see a fairly small overhead for using them, with potentially large gains in avoiding subtle modelling errors. Unit types will be available in MiniZinc in a near future release.

References

1 Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele. Object-oriented units of measurement. *SIGPLAN Not.*, 39(10):384–403, October 2004. `doi:10.1145/1035292.1029008`.

2 Astropy Collaboration. The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package. *The Astrophysical Journal*, 935(2), 2022.

3 Geoff Baldwin. Implementation of physical units. *SIGPLAN Not.*, 22(8):45–50, August 1987. `doi:10.1145/35596.35601`.

4 Christoph Berg. postgresql-unit, January 2023. URL: `https://github.com/mpusz/units`.

5 Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, and David L Woodruff. *Pyomo - optimization modeling in Python, 3rd Edition*, volume 67. Springer, 2021. `doi:10.1007/978-3-030-68928-5`.

6 Jip J. Dekker. MiniZinc Python, April 2023. URL: `https://github.com/MiniZinc/minizinc-python`.

7 A Dreiheller, B Mohr, and M Moerschbacher. Programming Pascal with physical units. *SIGPLAN Not.*, 21(12):114–123, December 1986. `doi:10.1145/15042.15048`.

8 N.H. Gehani. Databases and units of measure. *IEEE Transactions on Software Engineering*, SE-8(6):605–611, 1982. `doi:10.1109/TSE.1982.236021`.

9 Hernan E. Grecco. Pint, December 2023. URL: `https://github.com/mpusz/units`.

10 Michael Karr and David B. Loveman. Incorporation of units into programming languages. *Commun. ACM*, 21(5):385–391, May 1978. `doi:10.1145/359488.359501`.

11 Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994. `doi:10.1007/3-540-57880-3_23`.

12 Andrew John Kennedy. Programming languages and dimensions. Technical Report UCAM-CL-TR-391, University of Cambridge, Computer Laboratory, April 1996. `doi:10.48456/tr-391`.

13 R. Männer. Strong typing and physical units. *SIGPLAN Not.*, 21(3):11–20, March 1986. `doi:10.1145/382280.382281`.

14 Steve McKeever, Oscar Bennich-Björkman, and Omar-Alfred Salah. Unit of measurement libraries, their popularity and suitability. *Softw. Pract. Exp.*, 51(4):711–734, 2021. `doi:10.1002/SPE.2926`.

15 Mateusz Pusz, Johel Ernesto Guerrero Peña, Chip Hogg, and The mp-units project team. mp-units, May 2021. URL: `https://github.com/mpusz/units`.

16 Boris Schäling. *The Boost C++ libraries*. Boris Schäling, 2011.

A MiniZinc Unit Types Syntax

Unit types add syntax to MiniZinc to declare dimensions, dimension expression, basic unit types, and unit type expressions. The extensions are defined below.

Basic Dimension Declaration	*d*	::=	`unit type` *id* `;`	
Dimension Expression	*de*	::=	*id*	basic
		|	`coord(` *id* `)`	coordinate dimension
		|	*de* `*` *de*	multiplication
		|	*de* `/` *de*	division
		|	`1 /` *de*	inversion
		|	*de* `^` *k*	power (*k* is integer)
Dimension Declaration	*cd*	::=	`unit type` *id* `=` *de*`;`	
Basic Unit Type	*b*	::=	`unit` *id* `:` *id* `;`	declared
		|	`unit` *id* `:` *id* = *k* `@` *id* `;`	derived (*k* is arithmetic constant)
Unit Type Expression	*ue*	::=	*id*	basic
		|	`coord(` *id* `)`	coordinate type
		|	*ue* `*` *ue*	multiplication
		|	*ue* `/` *ue*	division
		|	`1 /` *ue*	inversion
		|	*ue* `^` *k*	power (*k* is integer)

The extension to the existing MiniZinc syntax is adding unit type annotations using `@`, together with the syntax for fine-grained unit iterators. Note that the `@` binds tighter than all arithmetic operators.

Expression	e	::=	k `@` ue	adding unit type (k is arithmetic)
Par/Var Declaration	dec	::=	[`var`] t [`@` ue] `:` id `;`	unit typed var/par
Index Set	is	::=	[id `of`] $enum$	fine-grained unit id

B MiniZinc Unit Typing Formalization

We now elaborate on the formal definitions in the paper and provide a full formalization of the unit typing in MiniZinc. We begin by formalizing unit types, and bottom up the type inference/checking procedure. We then show how we can add automatic coercions to programs that use multiple units for the same dimension.

B.1 Typing Rules

We use $\bot$ to represent a type error. A model contains m different dimensions $di_1, \ldots, di_m$ defined by the dimensions declared (or imported) into the model, and a dimension for each enumerated type defined in the model. We assume a set B of defined and declared basic unit types, together with each enumerated type defined in the model, and each fine grained unit expression. For each $b \in B$, $dim(b) = di_i$ for some i. We pick a *default* basic unit for each dimension di_i denoted $defb_i$ simply for notational convenience.

A *compound unit type* u is of the form $b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$ where each $n_i, 1 \leq i \leq m$ is an integer, and $dim(b_i) = di_i, 1 \leq i \leq m$. The dimension of u is $di_1^{n_1} di_2^{n_2} \cdots di_m^{n_m}$.

A *coordinate unit type* $coord(b)$ is defined only for declared or derived unit types $b \in B$. We let $dim(coord(b)) = coord(dim(b))$. We extend the meet operation to coordinate types in the obvious way $\sqcap(coord(b_1), u) = coord(\sqcap(b_1, b_2))$ iff $u = coord(b_2)$ and $\sqcap(b_1, b_2) \neq \bot$ and $\bot$ otherwise. We extend downcasting to coordinate types also, so $\downarrow(coord(b_1), u) = coord(\downarrow(b_1, b_2))$ if $u = coord(b_2)$ and $\bot$ otherwise. We extend multiplication of unit types to coordinate types as follows: $u \otimes coord(b)$ is $coord(b)$ iff $u = \mathbf{1}$ and $\bot$ otherwise. Note these extension definitions implicitly rely on the fact that each operations is commutative.

A *unit type* is either a proper unit type or a coordinate unit type.

Type expression evaluation is defined as follows.

$$
\begin{array}{rcl}
\tau(\mathbf{1}) & = & defb_1^0 \, defb_2^0 \cdots defb_m^0 \\
\tau(b) & = & b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m} \text{ where if } dim(b) = di_i,\ b_i = b, n_i = 1. \\
 & & \text{and } b_j = defb_j, n_j = 0, 1 \leq j \neq i \leq m \\
\tau(\texttt{coord}(u)) & = & \left\{ \begin{array}{ll} coord(\tau(u)), & \tau(u) = b_i, b_i \in B \\ \bot, & \text{otherwise} \end{array} \right. \\
\tau(u \times u') & = & \tau(u) \otimes \tau(u') \\
\tau(\mathbf{1}/u) & = & \left\{ \begin{array}{ll} b_1^{-n_1} b_2^{-n_2} \cdots b_m^{-n_m} & \tau(u) = b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m} \\ \bot & \text{otherwise} \end{array} \right. \\
\tau(u/u') & = & \tau(u) \otimes \tau(\mathbf{1}/u') \\
\tau(u\hat{\ }c) & = & \left\{ \begin{array}{ll} b_1^{n_1 \times c} b_2^{n_2 \times c} \cdots b_m^{n_m \times c}, & \tau(u) = b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}, c \text{ integer constant} \\ \bot, & \text{otherwise} \end{array} \right. \\
\tau(u) & = & \tau(e),\ u \text{ is a compound unit type defined by } u = e
\end{array}
$$

The unit typing rules for arithmetic expressions are defined below. Note that if in typing expression e some subexpression results in $\bot$ the whole expression is ill-typed, and we emit an error message.

$$
\begin{array}{rcl}
\mathit{type}(k) & = & \tau(\mathbf{1}),\ k \text{ is a numeric constant} \\
\mathit{type}(e@u) & = & \begin{cases} \tau(u), & \mathit{type}(e) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(x) & = & \tau(u),\ \text{identifier } x \text{ has declared unit type } u \\
\mathit{type}(\texttt{bool2int}(e)) & = & \tau(\mathbf{1}) \\
\mathit{type}(e_1 + e_2) & = & \begin{cases} \mathit{type}(e_1), & \mathit{type}(e_1) = \mathit{coord}(\mathit{type}(e_2)) \\ \sqcap(\mathit{type}(e_1), \mathit{type}(e_2)), & \text{otherwise} \end{cases} \\
\mathit{type}(e_1 - e_2) & = & \begin{cases} u, & \mathit{type}(e_1) = \mathit{type}(e_2) = \mathit{coord}(u) \\ \mathit{type}(e_1), & \mathit{type}(e_1) = \mathit{coord}(\mathit{type}(e_2)) \\ \sqcap(\mathit{type}(e_1), \mathit{type}(e_2)), & \text{otherwise} \end{cases} \\
\mathit{type}(e_1 \times e_2) & = & \tau(\mathit{type}(e_1)) \otimes \mathit{type}(e_2)) \\
\mathit{type}(e_1 / e_2) & = & \tau(\mathit{type}(e_1)/\mathit{type}(e_2)) \\
\mathit{type}(e_1 \texttt{ div } e_2) & = & \tau(\mathit{type}(e_1)/\mathit{type}(e_2)) \\
\mathit{type}(e_1 \texttt{ mod } e_2) & = & \begin{cases} \mathit{type}(e_1) & \mathit{type}(e_2) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(\texttt{abs}(e)) & = & \mathit{type}(e) \\
\mathit{type}(\texttt{max}(e_1, e_2)) & = & \sqcap(\mathit{type}(e_1), \mathit{type}(e_2)) \\
\mathit{type}(\texttt{min}(e_1, e_2)) & = & \sqcap(\mathit{type}(e_1), \mathit{type}(e_2)) \\
\mathit{type}(e_1 \mathbin{\hat{}} e_2) & = & \begin{cases} \tau(\mathit{type}(e_1)\hat{}e_2), & e_2 \text{ is a integer constant} \\ \tau(\mathbf{1}), & \mathit{type}(e_1) = \mathit{type}(e_2) = \tau(\mathbf{1}) \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(\mathit{trig}(e)) & = & \begin{cases} \tau(1), & \mathit{type}(e) = \tau(1) \text{ or } \mathit{type}(e) = \tau(b) \text{ where } \mathit{dim}(b) = \texttt{ angle} \\ \bot, & \text{otherwise} \end{cases} \\
 & & \text{where } \mathit{trig} \text{ is a trigonometric function } \{\texttt{sin}, \texttt{cos}, \ldots\}. \\
\mathit{type}(\mathit{upcast}(u, e)) & = & \begin{cases} \mathit{type}(u), & \sqcap(\mathit{type}(u), \mathit{type}(e)) = \mathit{type}(e) \\ \bot & \text{otherwise} \end{cases} \\
 & & \mathit{upcast} \text{ is a upcast function } \texttt{ceil}, \texttt{floor}, \texttt{round}. \\
\mathit{type}\begin{pmatrix} \texttt{if } e_1 \\ \texttt{then } e_2 \\ \texttt{else } e_3 \texttt{ endif} \end{pmatrix} & = & \begin{cases} \sqcap(\mathit{type}(e_2), \mathit{type}(e_3)), & \mathit{type}(e_1) = \texttt{bool} \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Container types in MiniZinc can contain objects with unit types. We extended the unit types to container types in the obvious way. We can define $\texttt{array}[\mathit{is}]\ \texttt{of}\ \ u$ an array with index set is of objects with unit type u;[7] $\texttt{set of}\ \ u$ a set of objects with unit type u; similarly for records and tuples. We extend the meet operation to apply to a sequence of types in the obvious manner $\sqcap([u_1, \ldots, u_n]) = \sqcap(u_1, \sqcap(u_2, \ldots, \sqcap(u_{n-1}, u_n) \cdots))$.

Unit typing rules for (some) containers are given below:

$$
\begin{array}{rcl}
\mathit{type}([e_1, \ldots, e_n]) & = & \begin{cases} \texttt{array[1..n] of } t & t = \sqcap([\mathit{type}(e_1), \cdots, \mathit{type}(e_n)], t \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(\{e_1, \ldots, e_n\}) & = & \begin{cases} \texttt{set of } t & t = \sqcap([\mathit{type}(e_1), \cdots, \mathit{type}(e_n)], t \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(e_1 \in e_2) & = & \begin{cases} \texttt{bool} & \mathit{type}(e_1) = u, \mathit{type}(e_2) = \texttt{set of } u', \sqcap(u, u') \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
\mathit{type}(e_1[e_2]) & = & \begin{cases} \mathit{type}(u) & \mathit{type}(e_1) = \texttt{array}[\mathit{int}]\texttt{ of } u \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

Unit typing rules for arithmetic constraints are defined below.

$$
\begin{array}{rcl}
\mathit{type}(e_1\ \mathit{op}\ e_2) & = & \begin{cases} \texttt{bool}, & \sqcap(\mathit{type}(e_1), \mathit{type}(e_2)) \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
 & & \mathit{op} \text{ is a mathematical relation } \texttt{=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}, \texttt{!=}.
\end{array}
$$

Unit typing rules for assignment are defined as:

$$
\mathit{type}(x = e) \quad = \quad \begin{cases} \texttt{bool} & \sqcap(\mathit{type}(x), \mathit{type}(e)) = \mathit{type}(x) \\ \bot & \text{otherwise} \end{cases}
$$

which requires the type of x to be a smaller unit type than that of e.

[7] Just showing the case for one dimensional arrays for simplicity.

Unit typing rules for predicates and user-defined functions are given below. Here, because the declared types can be parametric in unit (and other) types, we have to find a consistent mapping θ from type variables to fixed types that agrees with the types (including units) of the arguments.

$$\begin{array}{rcl}
\mathit{type}(p(e_1,\ldots,e_n)) & = & \left\{ \begin{array}{ll} \texttt{bool}, & \mathit{type}(e_i) = \tau(\theta(u_i)), 1 \leq i \leq n \text{ for type substitution } \theta \\ \perp, & \text{otherwise} \end{array} \right. \\
 & & \text{where } f \text{ is a predicate with unit type } u_1 \times \cdots \times u_n \rightarrow \texttt{bool} \\
\mathit{type}(f(e_1,\ldots,e_n)) & = & \left\{ \begin{array}{ll} \mathit{type}(\theta(u)), & \mathit{type}(e_i) = \tau(\theta(u_i)), 1 \leq i \leq n \text{ for type substitution } \theta \\ \perp, & \text{otherwise} \end{array} \right. \\
 & & \text{where } f \text{ is a user defined function with unit type } u_1 \times \cdots \times u_n \rightarrow u
\end{array}$$

A *fine-grained unit type* is written `array[`i `of` *enum*`] of` $u(i)$ where *enum* is an enumerated type, and $u(i)$ is a unit type expression making use of the identifier i.[8] To allow fine-grained unit types, we have to extend the type system to include unit type identifiers i which are replaced by the appropriate expression during type checking. There are treated as if they were a new basic type (so B includes all the unit type identifiers used in the model). Fine-grained types are introduced via variable declarations:

`array[`i `of` *enum*`] of` $u(i) : x$

declares a variable x to have (fine-grained) unit type `array[`i `of` *enum*`] of` $u(i)$. We extend the typing rules for array lookups to:

$$\begin{array}{rcl}
\tau(u) & = & u, \; u \text{ is a unit type variable.} \\
\mathit{type}(e_1[e_2]) & = & \left\{ \begin{array}{ll} \mathit{type}(u) & \mathit{type}(e_1) = \texttt{array}[\mathit{int}] \texttt{ of } u \\ \mathit{type}(u(e_2)) & \mathit{type}(e_1) = \texttt{array}[i \texttt{ of } \mathit{enum}] \texttt{ of } u(i), e_2 \text{ is an identifier} \\ \perp, & \text{otherwise} \end{array} \right.
\end{array}$$

Fine-grained unit type checking does not otherwise change the typing rules. Type equivalence relies on the unit type identifiers appearing identically in the two types, which means that all the identifiers must appear identically in each term.

[8] We restrict ourselves to one dimensional fine-grained types for simplicity, the extension to more dimensions is reasonably straightforward.