Unit Types for MiniZinc

Jip J. Dekker, **Jason Nguyen**,
Peter J. Stuckey, and Guido Tack

https://doi.org/10.4230/LIPIcs.CP.2025.10

MONASH University

OPTiMA
ARC TRAINING CENTRE IN
OPTIMISATION TECHNOLOGIES
INTEGRATED METHODOLOGIES
AND APPLICATIONS

Introduction

Consider this Knapsack problem

```
int: k;         % number of products to choose
int: limit;     % available weight limit
enum PRODUCT; % set of products available
array[PRODUCT] of int: weight;
array[PRODUCT] of int: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*profit[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Introduction

Type checks correctly, runs, but gives nonsense solutions

```
int: k;        % number of products to choose
int: limit;    % available weight limit
enum PRODUCT; % set of products available
array[PRODUCT] of int: weight;
array[PRODUCT] of int: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*profit[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Introduction

Type checks correctly, runs, but gives nonsense solutions

```
int: k;         % number of products to choose
int: limit;     % available weight limit
enum PRODUCT;   % set of products available
array[PRODUCT] of int: weight;
array[PRODUCT] of int: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*profit[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Should be weight

Introduction

- Model contains a ***unit error*** (but not a type error)
  - Comparing two integers with different meanings
    - weight (kg) vs profit ($)

Introduction

- Model contains a ***unit error*** (but not a type error)
  - Comparing two integers with different meanings
    - weight (kg) vs profit ($)
- Debugging models is difficult, so the more errors we can detect during compilation, the better

Introduction

- Model contains a ***unit error*** (but not a type error)
  - Comparing two integers with different meanings
    - weight (kg) vs profit ($)
- Debugging models is difficult, so the more errors we can detect during compilation, the better
- But we don't want to sacrifice runtime (solve time) performance, instead we must perform checking statically

Introduction

- Model contains a ***unit error*** (but not a type error)
  - Comparing two integers with different meanings
    - weight (kg) vs profit ($)
- Debugging models is difficult, so the more errors we can detect during compilation, the better
- But we don’t want to sacrifice runtime (solve time) performance, instead we must perform checking statically
- We also want to keep the system compatible with existing models

Dimensions and Units

- A dimension is a kind of measurement
  - distance, time, mass, worth, etc

Dimensions and Units

- A dimension is a kind of measurement
  - distance, time, mass, worth, etc

```
unit type distance;
unit type time;
% Can also create derived dimensions
unit type velocity = distance / time;
```

Dimensions and Units

- A basic unit has a particular dimension
  - km (distance), sec (time), kg (weight), dollars (worth)

Dimensions and Units

- A basic unit has a particular dimension
  - km (distance), sec (time), kg (weight), dollars (worth)

```
unit distance: m;
unit distance: km = 1000 @ m;    % derived unit
```

Dimensions and Units

- A basic unit has a particular dimension
  - km (distance), sec (time), kg (weight), dollars (worth)

```
unit distance: m;
unit distance: km = 1000 @ m;     % derived unit
```

```
unit time: sec;
unit time: minute = 60 @ sec;     % derived unit
unit time: hour = 60 @ minute;    % derived unit
```

Dimensions and Units

- A basic unit has a particular dimension
  - km (distance), sec (time), kg (weight), dollars (worth)

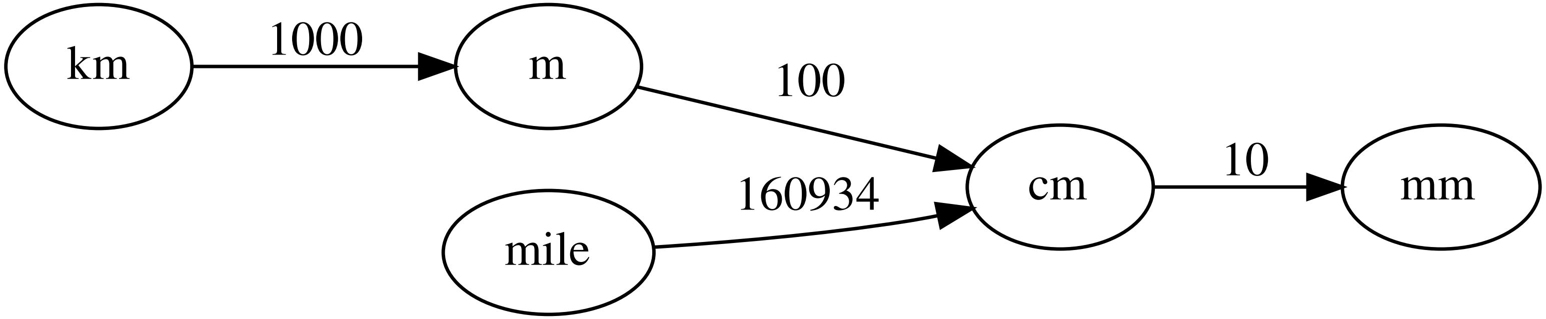
```
unit distance: m;
unit distance: km = 1000 @ m;    % derived unit
```

```
unit time: sec;
unit time: minute = 60 @ sec;    % derived unit
unit time: hour = 60 @ minute;   % derived unit
```

```
unit velocity: kmh = km / hour;  % derived unit
```

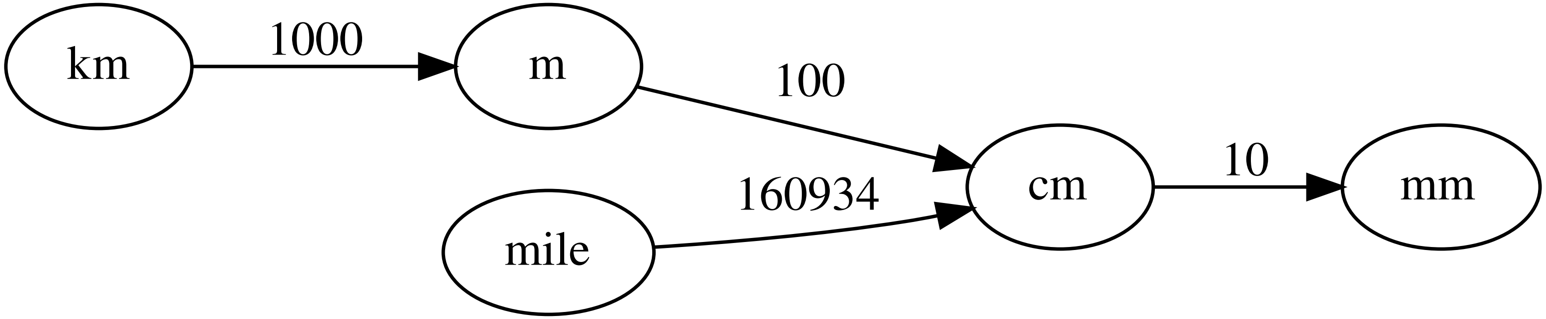
Dimensions and Units

- Basic units form graphs



Dimensions and Units

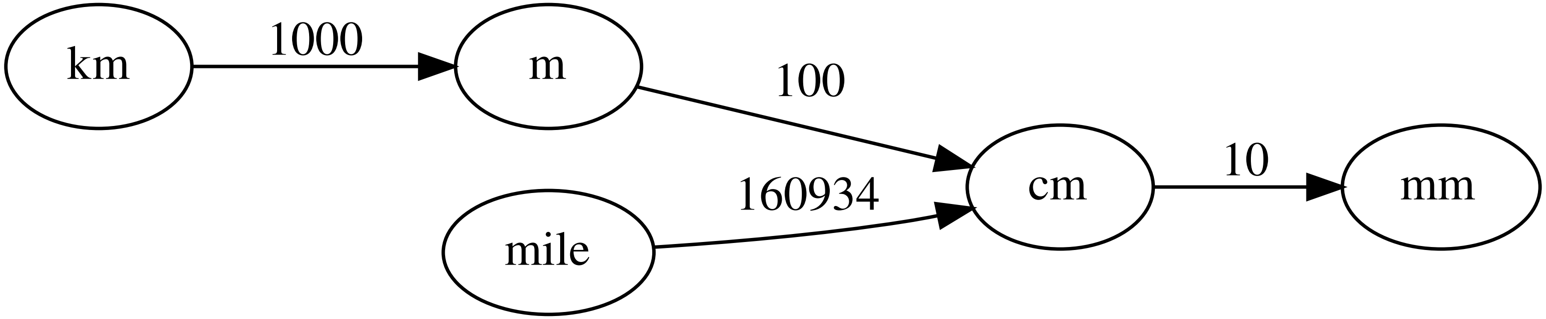
- Basic units form graphs



- To downcast from mile to mm, we need to multiply by 160934 × 10
  - $\downarrow(\text{mile}, \text{mm}) = 160934 \times 10$

Dimensions and Units

- Basic units form graphs



- To downcast from mile to mm, we need to multiply by 160934 × 10
  - $\downarrow$(mile, mm) = 160934 × 10
- The most general common unit of km and mile is cm
  - $\sqcap$(km, mile) = cm

The Unit Type System

- Numeric values are assigned complex units

  $$u \equiv b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$$

  where b_i is a basic unit of dimension i

  - A complex unit can only contain one basic unit of each dimension
  - Basic units with an exponent of zero may be omitted for brevity

The Unit Type System

- Numeric values are assigned complex units

  $$u \equiv b_1^{n_1} b_2^{n_2} \cdots b_m^{n_m}$$

  where b_i is a basic unit of dimension i
  - A complex unit can only contain one basic unit of each dimension
  - Basic units with an exponent of zero may be omitted for brevity
- We can extend the downcasting $\downarrow$ and meet $\sqcap$ operators to these complex units
  - $\downarrow$ fails if any basic unit downcast fails
  - $\sqcap$ fails if the exponents do not match, or if any basic unit meet fails

The Unit Type System

- We can multiply complex units $u \otimes v$
- The inverse of a unit $\mathbf{1} / u$ is found by negating its exponents

The Unit Type System

- We can multiply complex units $u \otimes v$
- The inverse of a unit $\boldsymbol{1} / u$ is found by negating its exponents
- The (most important) typing rules:
  - $type(k \mathbin{@} u) = u$
  - $type(e_1 + e_2) = \sqcap(type(e_1), type(e_2))$
  - $type(e_1 \times e_2) = type(e_2) \otimes type(\mathrm{e}_2)$
  - $type(e_1 \text{ div } e_2) = type(e_1) \otimes type(1 / e_2)$

Coercions

```
var int@kg: x;
var int@gram: y = x + 55@gram;   % automatic coercion
var int@kg: z = x + y;           % not allowed
```

- Automatic downcasting between units of a given dimension is possible as there is no loss of precision

Coercions

```
var int@kg: x;
var int@gram: y = x + 55@gram;   % automatic coercion
var int@kg: z = x + y;           % not allowed
```

- Automatic downcasting between units of a given dimension is possible as there is no loss of precision
- If we simply coerce all values to the finest unit, variable domains could end up very large, so we must be careful

Coercions

```
var int@kg: x;
var int@gram: y = x + 55@gram;   % automatic coercion
var int@kg: z = x + y;           % not allowed
```

- Automatic downcasting between units of a given dimension is possible as there is no loss of precision
- If we simply coerce all values to the finest unit, variable domains could end up very large, so we must be careful
- Automatic upcasts for integer variables are not allowed as they would require rounding

Fixing the Knapsack Problem

Applying unit types

```
int: k;              % number of products to choose
int@kg: limit;       % available weight limit
enum PRODUCT;        % set of products available
array[PRODUCT] of int@kg: weight;
array[PRODUCT] of int@dollar: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*profit[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Fixing the Knapsack Problem

Now the compiler can detect the error!

```
int: k;                 % number of products to choose
int@kg: limit;          % available weight limit
enum PRODUCT;           % set of products available
array[PRODUCT] of int@kg: weight;
array[PRODUCT] of int@dollar: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*profit[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

profit has unit **dollar**
limit has unit **kg**

Error found by compiler!

But we can do more!

(there are still plain, error-prone integers here)

Counting Types

Consider the correct model

```
int: k;               % number of products to choose
int@kg: limit;        % available weight limit
enum PRODUCT;         % set of products available
array[PRODUCT] of int@kg: weight;
array[PRODUCT] of int@dollar: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*weight[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Counting Types

Consider the correct model

```
int: k;              % number of products to choose
int@kg: limit;       % available weight limit
enum PRODUCT;        % set of products available
array[PRODUCT] of int@kg: weight;
array[PRODUCT] of int@dollar: profit;
array[PRODUCT] of var 0..infinity: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*weight[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

k and chosen are actually *counts* of PRODUCT

Counting Types

Enums are extended to create their own counting unit

```
int@PRODUCT: k; % number of products to choose
int@kg: limit;  % available weight limit
enum PRODUCT;   % set of products available
array[PRODUCT] of int@(kg/PRODUCT): weight;
array[PRODUCT] of int@(dollar/PRODUCT): profit;
array[PRODUCT] of var (0..infinity)@PRODUCT: chosen;

constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*weight[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Counting Types

Enums are extended to create their own counting unit

```
int@PRODUCT: k; % number of products to choose
int@kg: limit;  % available weight limit
enum PRODUCT;   % set of products available
array[PRODUCT] of int@(kg/PRODUCT): weight;
array[PRODUCT] of int@(dollar/PRODUCT): profit;
array[PRODUCT] of var (0..infinity)@PRODUCT: chosen;
```

Now even safer!

```
constraint sum(chosen) = k;
constraint sum(p in PRODUCT)(chosen[p]*weight[p]) <= limit;
solve maximize sum(p in PRODUCT)(chosen[p]*profit[p]);
```

Fine Counting Types

This model is unit correct

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

Fine Counting Types

This model is unit correct

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

But it contains a mistake!

Fine Counting Types

This model is unit correct

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

But it contains a mistake!

Should be p2

Fine Counting Types

What if we give each array element its own unit?

```
usage[r,p1]*chosen[p1]

+ usage[r,p2]*chosen[p1]

<= limit[r]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]

+ usage[Cost,Banana]*chosen[Apple]

<= limit[Cost]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost / Apple

```
+ usage[Cost,Banana]*chosen[Apple]
```

```
<= limit[Cost]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost / Apple × Apple

```
+ usage[Cost,Banana]*chosen[Apple]
```

```
<= limit[Cost]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost / Apple × Apple

```
+ usage[Cost,Banana]*chosen[Apple]
```

Cost / Banana

```
<= limit[Cost]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost / Apple × Apple

```
+ usage[Cost,Banana]*chosen[Apple]
```

Cost / Banana × Apple

```
<= limit[Cost]
```

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost / Apple × Apple

```
+ usage[Cost,Banana]*chosen[Apple]
```

Cost / Banana × Apple

```
<= limit[Cost]
```

Cost

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost

```
+ usage[Cost,Banana]*chosen[Apple]
```

Cost / Banana × Apple

```
<= limit[Cost]
```

Cost

Fine Counting Types

What if we give each array element its own unit?

```
usage[Cost,Apple]*chosen[Apple]
```

Cost

```
+ usage[Cost,Banana]*chosen[Apple]
```

Cost × Apple / Banana

```
<= limit[Cost]
```

Cost

Fine Counting Types

So we introduce fine counting types

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

Fine Counting Types

So we introduce fine counting types

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

Now we can detect the error!

Coordinate Types

Consider this excerpt of a scheduling problem

```
enum TASK;
array[TASK] of var int@minute: start;
array[TASK] of int@minute: duration;
constraint disjunctive(duration, start);
```

Coordinate Types

Consider this excerpt of a scheduling problem

```
enum TASK;
array[TASK] of var int@minute: start;
array[TASK] of int@minute: duration;
constraint disjunctive(duration, start);
```

⚠️ Arguments are flipped around

It's type correct, unit correct, and runs, but is wrong!

Coordinate Types

- Most numeric values in MiniZinc (and programming languages in general) are **differences**

Coordinate Types

- Most numeric values in MiniZinc (and programming languages in general) are **differences**
- We want to distinguish between delta (difference) unit types and absolute **coordinate** unit types

Coordinate Types

- Most numeric values in MiniZinc (and programming languages in general) are **differences**
- We want to distinguish between delta (difference) unit types and absolute **coordinate** unit types
- E.g. 25°C – 20°C = 5°C difference, but 25°C + 20°C makes no sense

Coordinate Types

- Most numeric values in MiniZinc (and programming languages in general) are **differences**
- We want to distinguish between delta (difference) unit types and absolute **coordinate** unit types
- E.g. 25°C – 20°C = 5°C difference, but 25°C + 20°C makes no sense
- We introduce coordinate unit types such that
  - $coord(x) + x = coord(x)$
  - $coord(x) - x = coord(x)$
  - $coord(x) - coord(x) = x$
  - And other arithmetic operations on $coord(x)$ are not allowed

Coordinate Types

Now using coordinate types

```
enum TASK;
array[TASK] of var int@coord(minute): start;
array[TASK] of int@minute: duration;
constraint disjunctive(duration, start);
```

Coordinate Types

Now using coordinate types

```
enum TASK;
array[TASK] of var int@coord(minute): start;
array[TASK] of int@minute: duration;
constraint disjunctive(duration, start);
```

Disjunctive now requires a coordinate unit as the first arument

Coordinate Types

Now using coordinate types

```
enum TASK;
array[TASK] of var int@coord(minute): start;
array[TASK] of int@minute: duration;
constraint disjunctive(duration, start);
```

⚠️ Disjunctive now requires a coordinate unit as the first arument

Now we can detect the error!

Global Constraints

- In order for units to catch more problems, we need to extend the global constraints to use them

Global Constraints

- In order for units to catch more problems, we need to extend the global constraints to use them
- We allow unit type parameters to appear in function parameters (and the return type)
  - $u stands in for any unit
  - $$E stands in for any enum type

Global Constraints

- In order for units to catch more problems, we need to extend the global constraints to use them
- We allow unit type parameters to appear in function parameters (and the return type)
  - $u stands in for any unit
  - $$E stands in for any enum type

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

Global Constraints

```
predicate cumulative(
   array[$$TASK] of var int@coord($time): start,
   array[$$TASK] of var int@$time: duration,
   array[$$TASK] of var int@$resource: usage,
   int@$resource: capacity
);
```

```
predicate span(
   var opt int@coord($time) start0,
   var int@time: duration0,
   array[$$TASK] of var opt int@coord($time): start,
   array[$$TASK] of var int@$time: duration
);
```

Global Constraints

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

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

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

Global Constraints

```
predicate knapsack(
   array [$$ITEM] of int@($WEIGHT/$$ITEM): weight,
   array [$$ITEM] of int@($PROFIT/$$ITEM): profit,
   array [$$ITEM] of var int@$$ITEM: chosen,
   var int@$WEIGHT: total_weight,
   var int@$PROFIT: total_profit
);
```

```
predicate knapsack(
   array [i of $$ITEM] of int@($WEIGHT/i): weight,
   array [i of $$ITEM] of int@($PROFIT/i): profit,
   array [i of $$ITEM] of var int@i: chosen,
   var int@$WEIGHT: total_weight,
   var int@$PROFIT: total_profit
);
```

Evaluation

- We examined the applicability of unit types to past MiniZinc Challenge problems (2021 – 2024)

Year	Units applicable	Mean size increase		Benchmarks using the unit type feature			
		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

Evaluation

- We examined the applicability of unit types to past MiniZinc Challenge problems (2021 – 2024)
- Unit types can be applied to most MiniZinc problems

Year	Units applicable	Mean size increase		Benchmarks using the unit type feature			
		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

Evaluation

- We examined the applicability of unit types to past MiniZinc Challenge problems (2021 – 2024)
- Unit types can be applied to most MiniZinc problems
- Unit types have no runtime performance impact

Year	Units applicable	Mean size increase		Benchmarks using the unit type feature			
		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

Evaluation

- We examined the applicability of unit types to past MiniZinc Challenge problems (2021 – 2024)
- Unit types can be applied to most MiniZinc problems
- Unit types have no runtime performance impact
- Written program size increase is minimal

		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

Conclusion

- Unit types provide more safety than strong typing alone

Conclusion

- Unit types provide more safety than strong typing alone
- The overhead of using unit types in other languages makes them less attractive

Conclusion

- Unit types provide more safety than strong typing alone
- The overhead of using unit types in other languages makes them less attractive
- There is a strong case for them in modelling languages as debugging is much more difficult

Conclusion

- Unit types provide more safety than strong typing alone
- The overhead of using unit types in other languages makes them less attractive
- There is a strong case for them in modelling languages as debugging is much more difficult
- Some units, such as counting types are specific to discrete optimisation

Conclusion

- Unit types provide more safety than strong typing alone
- The overhead of using unit types in other languages makes them less attractive
- There is a strong case for them in modelling languages as debugging is much more difficult
- Some units, such as counting types are specific to discrete optimisation
- The MiniZinc implementation of unit types provides extra safety with no runtime cost and minimal code overhead, while ensuring existing models continue to work

Try the prototype at

https://www.minizinc.org/unit-types