Converting MiniZinc to FlatZinc

Nicholas Nethercote

Version 0.8

1 Introduction

This document specifies how to convert MiniZinc to FlatZinc.

We will use the MiniZinc model and example data for a restricted job shop scheduling problem in Figures 1 and 2 as a running example.

This MiniZinc model instance is translated into the FlatZinc code shown in Figure 3. Line 30 is the original two-dimensional array of decision variables, mapped to a zero-indexed one-dimensional array. Lines 32–35 are variables introduced by Boolean decomposition. Lines 36–45 are the constraints. Lines 37 and 39 result from line 12, lines 38 and 40 result from line 13, and lines 41–46 result from lines 14–15 and 7–8.

2 The Translation

The translation has two parts: flattening, and the rest.

2.1 Flattening

Flattening involves the following simple steps that statically evaluate (or *reduce*) the model and data as much as possible. There is no fixed order to the steps because some enable others, which can then enable further application of previously applied steps. Therefore, they must be repeatedly applied, e.g. by iterating until a fixpoint is reached, or by re-flattening child nodes of expressions that have been flattened.

Parameter substitution. This step substitutes any atomic literal value assigned to a global or let-local scalar parameter throughout the model, and removes the declaration and assignment. For example, with `size = 2` we substitute `2` for `size`, but `size = 2 + y` would not be substituted until it is fully reduced. Once all the uses of a parameter have been replaced, its declaration and assignment can be removed.

```
XXX: need to give more examples throughout this doc, at least one per step.

XXX: type-inst constraints make things complicated.  Eg:

  1..3: x = 10;

This causes a flattening-time abort because 10 isn't within 1..3.  So
type-inst constraints must be checked before the substitution can occur.

XXX: Should we also substitute decision variables, if they are assigned to?
Doing so would avoid the creation of some equality constraints.  What about
arrays -- it can cause code bloat, but it might be important in some cases.
It affects whether assignment is the same as equality...
```

```
 0    % (square) job shop scheduling in MiniZinc
 1    int: size;                                  % size of problem
 2    array [1..size,1..size] of int: d;          % task durations
 3    int: total = sum(i,j in 1..size) (d[i,j]);  % total duration
 4    array [1..size,1..size] of var 0..total: s; % start times
 5    var 0..total: end;                          % total end time
 6
 7    predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
 8        s1 + d1 <= s2 \/ s2 + d2 <= s1;
 9
10    constraint
11        forall(i in 1..size) (
12            forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
13            s[i,size] + d[i,size] <= end /\
14            forall(j,k in 1..size where j < k) (
15                no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
16            )
17        );
18
19    solve minimize end;
```

Figure 1: MiniZinc model (`jobshop.mzn`) for the job shop problem.

```
20    size = 2;
21    d = [| 2,5
22         | 3,4 ];
```

Figure 2: MiniZinc data (`jobshop2x2.data`) for the job shop problem.

```
XXX: possible annotation behaviour.  If the assigned value (the RHS) is
annotated:
- the annotation should be copied.  Eg.

    x = y::z;
    ...x...                 --> ...y::z...

If the variable is annotated:
- Ignore it?  Eg:

    var int: x :: foo = 3;
    ...x...                 --> ...x...

  Putting an annotation on a scalar parameter like this doesn't seem very
  useful.  But if we allow decision variables to be involved with
  substitution, it becomes more complicated.  But anything other than
  ignoring doesn't seem sensible.

  [[Seb:]]
  Should this be the following instead?
    ...x...                 --> ...3...

  If we say that the "foo" annotation is on the declaration, then
```

```
30   array[0..3] of var 0..14: s;
31   var 0..14: end;
32   var bool: b1;
33   var bool: b2;
34   var bool: b3;
35   var bool: b4;
36   constraint  int_lin_le     ([1,-1], [s[0], s[1]], -2);
37   constraint  int_lin_le     ([1,-1], [s[1], end ], -5);
38   constraint  int_lin_le     ([1,-1], [s[2], s[3]], -3);
39   constraint  int_lin_le     ([1,-1], [s[3], end ], -4);
40   constraint  int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1);
41   constraint  int_lin_le_reif([1,-1], [s[2], s[0]], -3, b2);
42   constraint  bool_or(b1, b2, true);
43   constraint  int_lin_le_reif([1,-1], [s[1], s[3]], -5, b3);
44   constraint  int_lin_le_reif([1,-1], [s[3], s[1]], -4, b4);
45   constraint  bool_or(b3, b4, true);
46   solve minimize end;
```

Figure 3: FlatZinc translation of the MiniZinc job shop model.

```
  there is no reason it should be attached to all occurrences of x.
```

Built-ins evaluation. This step evaluates all built-ins with constant, atomic literal arguments. For example, `2-1` (from `size-1`, after parameter substitution) in our example becomes `1`.

```
XXX: possible annotation behaviour.  If the annotation is on the built-in
itself:
- ignore it?  Such annotations don't make much sense on par expressions.  Eg:

    (2 - 1)::x      --> 1

XXX: If the annotation is on an argument:
- ignore it?  Again, not very sensible?  Eg:

    (2::x - 1)      --> 1

  [[Seb:]]
  A principle that "annotations are deleted from par expressions" would
  cover these cases.
```

Comprehension unrolling. This step unrolls all set and array comprehensions, once the generator ranges are fully reduced, replacing the generator variables in the expressions with literal values.

Compound built-in unrolling. This step unrolls compound built-ins (such as `sum` and `forall`; the full list is in Section 3.1) by replacing them with multiple lower-level operations.

We will use lines 11, 14 and 15 of Figure 1 as the starting point of a running example. They unroll to give the following conjunction (the first conjunct has `i=1`, `j=1` and `k=2`; the second has `i=2`, `j=1` and `k=2`).

```
   no_overlap(s[1,1], d[1,1], s[2,1], d[2,1]) /\
```

```
   no_overlap(s[1,2], d[1,2], s[2,2], d[2,2])

XXX: annotations: any annotations are copied to all operations?  eg:

  sum([1, 2, 3])::x     --> (1 + (2 + 3)::x)::x

Or just the outermost one?  eg:

                        --> (1 + (2 + 3))::x

  [[Seb:  My choice; perhaps with additional special treatment of
  conjunction/forall.]]
```

Fixed array access replacement. This step replaces all array accesses involving fixed indices and fixed elements with the appropriate value. Once all the accesses of an array have been replaced, its declaration and assignment can be removed. For example, our running example becomes:

```
   no_overlap(s[1,1], 2, s[2,1], 3) /\
   no_overlap(s[1,2], 5, s[2,2], 4)
```

```
XXX: What about if the access has fixed indices but a non-fixed element?  Do
them too?
```

If-then-else evaluation. This step evaluates each if-then-else expression, once its condition is fully reduced. This is always possible because if-then-else conditions must be fixed.

Predicate inlining. This step replaces each call to a defined predicate with its body, substituting actual arguments for formal arguments. This is easy because predicates cannot be recursive, either directly or mutually. Calls to predicates lacking a definition (such as those in the MiniZinc globals library) are left as-is. For example, the first conjunct from our running example (after fixed array accesses to `d` have been flattened) becomes:

```
  s[1,1] + 2 <= s[2,1] \/ s[2,1] + 3 <= s[1,1]
```

Predicates can call other predicates, and the inlining can be done outside-in or inside-out. For example, in `f(g(x))` we can inline `f` first, then `g`, or vice versa.

2.2 Post-flattening

After flattening, we apply the following steps once each, in the given order. Those steps marked with a '*' can be performed in more than one way, so their output depends on the exact details of the implementation, although the results should not vary greatly.

Stand-alone assignment removal. This step removes each stand-alone assignment by merging it with the appropriate variable declaration.

Let floating. This step moves let-local decision variables to the top-level and renames them appropriately.

```
Be clear that the type-inst is preserved.
```

Boolean decomposition*. This step decomposes all Boolean expressions that are not top-level conjunctions. It replaces each sub-expression with a new Boolean variable (also adding a declaration for each variable), and adds conjuncts equating these new variables with the sub-expressions they replaced. This facilitates the later introduction of reified constraints.

```
XXX: new variables are annotated with var_is_introduced
```

For example, this expression:

```
s[1,1] + 2 <= s[2,1] \/ s[2,1] + 3 <= s[1,1]
```

becomes:

```
((b1 \/ b2) <-> true) /\
((s[1,1] + 2 <= s[2,1]) <-> b1) /\
((s[2,1] + 3 <= s[1,1]) <-> b2)
```

Declarations are also added for the new Boolean variables `b1` and `b2`.

Numeric decomposition*. This step decomposes numeric equations or inequations in a manner similar to Boolean decomposition, by renaming each non-linear sub-expression with a new variable.

Set decomposition*. This step decomposes compound set expressions into primitive set constraints in a manner similar to Boolean/numeric decomposition.

(In)equality normalisation*. This step normalises (in)equations, e.g. it converts `>=` into `<=`, moves sub-expressions so the right-hand side is constant, and replaces negations with multiplications by -1. This facilitates the later introduction of linear (in)equality constraints. For example, the second conjunct from our running example becomes:

```
(s[1,1] + (-1)*s[2,1] <= -2) <-> b1
```

An exception: if a solve item involves a linear expression, it should be preserved, so it can later be transformed into a call to `int_float_lin`.

Array simplification. This step simplifies all arrays to one-dimensional, zero-indexed arrays. It also updates any remaining array accesses accordingly. For example, our running example becomes:

```
(s[0] + (-1)*s[2] <= -2) <-> b1
```

```
XXX: be clearer:  affects both array variables and array literals?
```

Anonymous variable naming. This step replaces each anonymous variable (‘`_`’) with a newly introduced variable of the appropriate type.

```
XXX: introduced variables are annotated with var_is_introduced.
```

Conversion to FlatZinc built-ins. This step converts the remaining MiniZinc built-ins and array accesses into FlatZinc built-ins. The FlatZinc built-ins may be type-specialised, and will be reified if the MiniZinc built-in occurs inside a Boolean expression (other than conjunction).

The most complex cases involves linear (in)equalities. Each one is replaced with a (type-specific, possibly reified) linear predicate, unless it can be replaced with a simpler arithmetic constraint. For example, our running example becomes:

```
(s[0] + (-1)*s[2] <= -2) <-> b1
```

becomes the reified FlatZinc built-in:

```
int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1)
```

Similarly, a linear expression in a solve item can be converted to a call `int_float_lin`.

The next case is that array accesses involving non-fixed indices are replaced with element constraints.

After that, conjunctions and disjunctions of identifiers (such as those that might be produced by reification) can be replaced with the N-ary conjunction and disjunction constraints. For example:

```
constraint B1 \/ B2 \/ B3 \/ B4;
```

becomes:

```
constraint array_bool_or([B1, B2, B3, B4]);
```

This step is not strictly necessary as it can be emulated with binary conjunction and disjunction, but it may produce FlatZinc models that can be solved faster.

The remaining cases are simpler. For example, `(b1 \/ b2) <-> true` becomes `bool_or(b1, b2, true)`, and `x * y = z` becomes `int_times(x, y, z)`. Section 3 specifies them in detail.

Top-level conjunction splitting. This step splits top-level conjunctions into multiple constraint items. For example `constraint a /\ b;` becomes two items.

Annotation declaration removal. This step removes any annotation declarations.

2.3 Annotations

`XXX: this section can be removed once annotation behaviour is clearly specified for all steps.`

Most annotations are maintained during the translation. Fixed expressions within annotations are evaluated like other expressions. When annotated expressions are unrolled, the annotation is copied to the resulting operations. When expressions are reified, their annotations are lost.

`[[Seb:]] because ... ?`

Auxiliary variables introduced during translation are annotated with `var_is_introduced`.

`[[Jakob:]] From my experience with the ttt hybrid it would be good if annotations would propagate to all introduced constraints and variables. An example:`

`mzn:`

```
var int: x::this, y::this;
constraint (x*y <= 5)::this;

fzn:
var l..u: x::this;
var l..u: y::this;
var l..u: TIMES___1::var_is_introduced::this;

constraint int_times(x, y, TIMES___1)::this;
constraint int_le(TIMES___1, 5)::this;

[[Jakob:]] I can currently see four generic annotation propagation beahviours,
it would be nice if on could choose between different ways of annotation
propagation something like the following could be interesting:

prop_on_introduced_vars         annotation a1;
prop_on_introduced_constraints  annotation a2;
prop_on_introduced_vars_cons    annotation a3;
no_prop                         annotation a4; (default behaviour)
```

2.4 Variations

These transformations provide a translation that is standard while also supporting much of what a solver writer would want. They are clearly not ideal for every underlying solver. For example, solvers may be more efficient on the undecomposed versions of Boolean constraints or non-linear constraints. It would be good to be able to control which transformations should be applied for a particular solver once we have more experience.

3 MiniZinc Built-ins

This section gives full details on how the MiniZinc built-in operations are translated to FlatZinc.

3.1 Compound Built-ins

Compound built-ins are unrolled into sequences of simpler MiniZinc operations, which are then translated further.

- `sum` is unrolled with `+`.
- `product` is unrolled with `*`.
- `forall` is unrolled with `/\`.
- `exists` is unrolled with `\/`.
- `array_union` is unrolled with `union`.
- `array_intersect` is unrolled with `intersect`.

3.2 Atomic Built-ins

Atomic built-ins with a fixed return value are evaluated and replaced with a value. The remainder are replaced with FlatZinc built-ins according to the list below. The comment after each type-inst signature indicates briefly how it is handled ("`--> x`" indicates that a built-in is renamed to "`x`"). Some of them will be converted to the reified version if they occur inside a Boolean expression other than conjunction.

```
var bool: '<'(var int,        var int)        % --> int_lt
var bool: '<'(var float,      var float)      % --> float_lt
var bool: '<'(var bool,       var bool)       % --> bool_lt
var bool: '<'(var set of int, var set of int) % --> set_lt
% Similarly:
%   '>'      --> *_gt
%   '>='     --> *_ge
%   '<='     --> *_le
%   '=='/'=' --> *_eq
%   '!='     --> *_ne

var int:   '+'(var int,   var int)        % --> int_plus    or int_lin_*
var float: '+'(var float, var float)      % --> float_plus  or float_lin_*
var int:   '-'(var int,   var int)        % --> int_minus   or int_lin_*
var float: '-'(var float, var float)      % --> float_minus or float_lin_*
var int:   '*'(var int,   var int)        % --> int_times   or int_lin_*
var float: '*'(var float, var float)      % --> float_times or float_lin_*

var int:   '+'(var int  )                 % removed
var float: '+'(var float)                 % removed
var int:   '-'(var int  )                 % --> int_negate
var float: '-'(var float)                 % --> float_negate

var int:   'div'(var int,   var int)      % --> int_div
var int:   'mod'(var int,   var int)      % --> int_mod
var float: '/'  (var float, var float)    % --> float_div

var int:   min(var int,   var int)        % --> int_min
var float: min(var float, var float)      % --> float_min
var int:   max(var int,   var int)        % --> int_max
var float: max(var float, var float)      % --> float_max

var int:   abs(var int  )                 % --> int_abs
var float: abs(var float)                 % --> float_abs

var bool: '/\' (var bool, var bool)       % becomes two constraint items
var bool: '\/' (var bool, var bool)       % --> bool_or
var bool: '->' (var bool, var bool)       % --> bool_right_imp
var bool: '<-' (var bool, var bool)       % --> bool_left_imp
var bool: '<->'(var bool, var bool)       % --> bool_eq
var bool: 'xor'(var bool, var bool)       % --> bool_xor

var bool: 'not'(var bool)                 % --> bool_not

var bool: 'in'(var int,  var set of int)          % --> set_in

var bool: 'subset'  (var set of int, var set of int)  % --> set_subset
```

```
var bool: 'superset'(var set of int, var set of int)  % --> set_subset

var set of int: 'union'    (var set of int,
                            var set of int)       % --> set_union
var set of int: 'intersect'(var set of int,
                            var set of int)       % --> set_intersect
var set of int: 'diff'     (var set of int,
                            var set of int)       % --> set_diff
var set of int: 'symdiff'  (var set of int,
                            var set of int)       % --> set_symdiff

var int: card(var set of int)                     % --> set_card

string: show(_)               % --> show (i.e. unchanged)
string: show_cond(_, _, _)    % --> show_cond (i.e. unchanged)
string: '++'(string, string)  % --> a comma in an array literal, e.g.
                              %         ["x = " ++ show(x)]
                              %     --> ["x = ",   show(x)]
```

3.3 Array Accesses

Array accesses with *var* indices are converted to element constraints. Here, on the left-hand side, we represent the type-inst signature of array accesses with the name `'[]'`, but this is not valid MiniZinc.

```
'[]'(array[int] of     bool,  var int) -> var bool
                                          % --> array_bool_element
'[]'(array[int] of var bool,  var int) -> var bool
                                          % --> array_var_bool_element
'[]'(array[int] of     int,   var int) -> var int
                                          % --> array_int_element
'[]'(array[int] of var int,   var int) -> var int
                                          % --> array_var_int_element
'[]'(array[int] of     float, var int) -> var float
                                          % --> array_float_element
'[]'(array[int] of var float, var int) -> var float
                                          % --> array_var_float_element
'[]'(array[int] of     set of int, var int) -> var set of int
                                          % --> array_set_element
'[]'(array[int] of var set of int, var int) -> var set of int
                                          % --> array_var_set_element
```

4 Syntax Differences Between the Languages

FlatZinc's syntax is mostly a subset of MiniZinc's, in the sense that any syntactically-valid FlatZinc model is a syntactically-valid MiniZinc model. (The exception to this is that output items have a slightly different syntax in the two languages.)

Nonetheless, there are some subtleties. The following list covers some potential pitfalls that may trip someone writing a MiniZinc-to-FlatZinc translator.

- MiniZinc items can appear in any order. FlatZinc items can not: variable declarations items must precede constraint items, which must precede the solve item, which must precede (if present) the output item.

- In FlatZinc, call expressions (e.g. `int_lt(a, 3)`) can only appear at the top level of constraint items. They cannot be used as sub-expressions. (Hence the "Flat" in "FlatZinc".)
- In MiniZinc, objective functions in solve items can be arbitrary expressions. In FlatZinc they must be identifiers; an arbitrary expression can be used as an objective function by assigning it to a variable.
- In MiniZinc, in all lists, the comma or semi-colon separator can also be used as a terminator. For example, in a function call `foo(a,b,c)` is equivalent to `foo(a,b,c,)`. This is so that multi-line lists can have a separator after every line, which can make code more consistent and easier to edit. For example:

  ```
  set of int: s = {
      1,
      2,
      3,
  };
  ```

  FlatZinc does not allow this; commas are always separators, and `foo(a,b,c,)` is a syntax error.