FlatZinc 0.9 to 1.0 Transition Guide

Julien Fischer

1 Introduction

This document is a guide to the changes introduced in version 1.0 of FlatZinc (and where relevant MiniZinc). It is intended to assist the authors of FlatZinc implementations in updating their implementations. References to the "Specification of FlatZinc" are enclosed in parentheses.

The new language features are also discussed in the "FlatZinc Summary" document.

2 Output Items

Output items (FlatZinc 0.9, section 7.4), have been completely removed from the FlatZinc language. Section 4 describes how output is now handled in FlatZinc 1.0.

Rationale. *The solver should not have to bother about complex output. By making the solver output simple, a post-process can recover the complex output.*

3 Built-in string operations

The built-in string operations `show/1` and `show_cond/3` (FlatZinc 0.9, appendix A.7) have been removed from the FlatZinc language.

Rationale. *With the removal of output items these operations are now unnecessary.*

4 Output in FlatZinc 1.0

This section describes how output is now handled in FlatZinc 1.0. This is the largest change from FlatZinc version 0.9.

```
var 1..10: x :: output_var;
array[1..4] of var 1..10: ys :: output_array([0..1, 0..1]);
var 1..10: z;
solve :: int_search(ys, input_order, indomain_min, complete) satisfy;
```

might result in the following being printed to the *standard output* when evaluated:

```
x = 1;
ys = array2d(0..1, 0..1, [1, 1, 1, 1]);
```

1. The annotation `output_var` attached to a non-array variable means that the value of that variable should be printed *on a single line* as:

   ```
   name = value;
   ```

when a solution is found.

2. The annotation `output_array([`*`IndexRange1`*`, ..., `*`IndexRangeN`*`])` attached to an array variable means that that variable should be printed *on a single line* as:

   ```
   name = arrayNd(IndexRange1, ..., IndexRangeN, [<ArrayValues>]);
   ```

3. Output variables (those annotated with either `output_var` or `output_array`) are printed in ascending lexicographic ASCII order.

4. Variables that do not have a `output_var` or `output_array` annotation are not printed at all. Parameters are never included in the output.

5. The output corresponding to each output variable must appear on a separate line.

6. The specification of how FlatZinc values should be printed is now more precise (FlatZinc 1.0, Section 9.4.1). Part of the conformance test suite checks this.

7. A solution is terminated by ten consecutive dashes, `----------`, on a line by themselves.

8. If a solver finishes searching the entire search tree then it should print ten consecutive equals signs, `==========`, on a line by themselves. If the output consists solely of this line of equals signs then solver was unable to find any solution. Previous versions of FlatZinc required the implementation to print `No solution` as this point.

9. Any other output should appear as a FlatZinc comment on a line by itself. The intention is that such comments be used for reporting any statistics that may have been gathered by the solve. For example:

   ```
   x = 1;
   y = array2d(1..2, 1..2, [1, 2, 3, 4]);
   ----------
   % Number of choicepoints: 4001
   % Total memory used: 4Gb
   ```

Rationale. *The new output specification ensures that individual solutions can be used as input to a MiniZinc model. This is useful, for example, when checking that the solution does actually satisfy the constraints.*

5 Predicate Declarations

A FlatZinc model instance may now contain declarations for non-standard built-in constraints. Such declarations, if present must occur before any variable declarations. For example:

```
predicate all_different(array[int] of var int: xs);
```

Rationale. *Adding predicate declarations makes it possible to type-check model instances in a self-contained way. Whilst this is not typically important for FlatZinc solver implementations, it can be important for tools that transform FlatZinc, for example with the FlatZinc-to-XML convert included with version 1.0 for the G12 MiniZinc distribution.*

- Most FlatZinc implementations can simply ignore predicate declarations.
- Array types in FlatZinc predicate declarations do not have to fix the array's size. Predicate declarations are the *only* place in FlatZinc where the size of an array need not be specified.
- `mzn2fzn` 1.0 has a command line option `--no-output-pred-decls`, that causes it not to emit predicate declarations.
- The argument types of non-standard built-in constraints must be valid FlatZinc types.

6 Functional relations

Two new annotations allow functional relations between variables to be expressed. The annotation `is_defined_var` is used to identify variables that are defined by some constraint as a function of other variables. The annotation `defines_var(x)` is used to identify the constraint that functionally defines the variable `x`. For example, the following fragment of FlatZinc defines `z` to be the sum of the variables `x` and `y`:

```
var int: x;
var int: y;
var int: z :: is_defined_var;

constraint int_plus(x, y, z) :: defines(z);
```

Rationale. *Some solvers may require this information or maybe be able to take advantage of it should it be available.*

- `mzn2fzn` 1.0 now introduces `is_defined_var` and `defined_var` annotations where appropriate.

7 Changes to standard solve annotations

The standard solve annotations (FlatZinc 1.0, appendix B.2) now use nested annotations as their arguments rather than strings.

The following is a solve annotation in FlatZinc 0.9:

```
solve :: int_search([x, y, z], "input_order", "indomain", "complete")
    satisfy;
```

The equivalent solve item in FlatZinc 1.0 is:

```
solve :: int_search([x, y, z], input_order, indomain, complete)
   satisfy;
```

Rationale. *This change allows tools that manipulate MiniZinc to perform better semantic checks of search annotations.*

- This change makes use of nested annotations, which were introduced in FlatZinc 0.9.

8 Sequential search

A new annotation has been introduced to force a sequential order on search annotations.

```
solve :: seq_search([
            int_search(xs, first_fail, indomain_min, complete),
            set_search(ys, input_order, indomain_max, complete)
        ]) satisfy;
```

The `seq_search` annotation in the above example forces the `int_search` to be carried out before the `set_search`.

Rationale. *Top-level annotations on an item are unordered. There was no existing mechanism for ensuring that if there were multiple search annotations that they would be carried out in a particular order.*

- `seq_search` annotations may be nested.

9 MiniZinc global constraint management

Previously all of the global constraint definitions were provided in a file named `globals.mzn`. FlatZinc implementors had to tailor this file to their specific solver, either by providing alternative decompositions or marking specific constraints as built-ins. The latter is done by providing a "bodyless" predicate definition for the constraint.

We have changed the way in which global constraints are handled in version 1.0 so that FlatZinc implementors need only provide alternative global constraint definitions for those specific constraint that they wish to redefine. In particular, the new approach avoids having to duplicate large chunks of `globals.mzn` and the maintenance problems that doing so entailed.

In version 1.0, `globals.mzn`, has been split into pieces, with at least one file per global constraint (the significance of the "at least one" is explained below). `globals.mzn` now consists solely of include items.

Note that from the users' perspective nothing has changed. Users can continue to include `globals.mzn` in their models and use the global constraints as in previous versions.

9.1 Structure of the MiniZinc global constraint library

The directory structure of the MiniZinc globals library version 1.0 is as follows:

```
lib/minizinc/
lib/minizinc/std/
lib/minizinc/g12_fd/
lib/minizinc/g12_lazyfd/
```

(The paths above are relative to the directory in which the MiniZinc distribution is installed.)

The subdirectory `std` contains the default decompositions of the global constraints, each in a separate file or files, and `globals.mzn`. This directory is always the last directory in which `mzn2fzn` will search for included files.

The directories `g12_fd` and `g12_lazyfd` contain solver-specific global constraint definitions for the G12/FD and G12/LazyFD solvers respectively.

We can instruct `mzn2fzn` to use the definitions in, for example the `g12_fd` subdirectory, in preference to those in the `std` subdirectory by using the new `--globals-dir` option. (`-G` may be used as a synonym for `--globals-dir`.)

For example:

```
$ mzn2fzn -G g12_fd foo.mzn
```

will cause `mzn2fzn` to search for files in the directory `lib/minizinc/g12_fd`, before searching in `lib/minizinc/std`. As with previous versions, `mzn2fzn` will use the *first* instance of an included file it finds.

The intention is that FlatZinc implementators should install any files containing solver-specific global constraint definitions in a subdirectory under `lib/minizinc`.

With the changes described above the directory search order for `mzn2fzn` is now:

1. The current working directory.
2. Any directory specified using the `--search-dir` or `-I` option.
3. Any directory specified using the `--globals-dir` or `-G` option.
4. `lib/minzinc/std`

Directories specified using `--search-dir` are searched in the order in which they are given on the command line. Similarly, for `--globals-dir`.

9.2 Specialising global constraints

Each non-overloaded MiniZinc global constraint is defined in its own `.mzn` file.

The definition of an overloaded MiniZinc global constraint is split between several file: one file per type signature that contains a *non-overloaded* auxiliary definiton of the constraint and an overall file that combines these auxiliary versions into the overloaded definition.

We shall illustrate this using the default definition of the `all_different` constraint. This involves the following files:

```
lib/minizinc/std/all_different.mzn
lib/minizinc/std/all_different_int.mzn
lib/minizinc/std/all_different_set.mzn
```

The file `all_different_int` contains an integer version of the `all_different` constraint named `all_different_int`. The contents of the file are as follows:

```
predicate all_different_int(array[int] of var int: x) =
    forall(i,j in index_set(x) where i < j) ( x[i] != x[j] );
```

Likewise, `all_different_set.mzn` contains the `all_different_set.mzn`.

These two files are combined in the file `all_different.mzn` to create the overloaded version of `all_different` as follows:

```
include "all_different_int.mzn";
include "all_different_set.mzn";

predicate all_different(array[int] of var int: x) =
    all_different_int(x);

predicate all_different(array[int] of var set of int: x) =
    all_different_set(x);
```

The reason for the above treatment of overloaded global constraints is twofold: firstly to minimise the amount of work that needs to be done specialise (parts of) the constraint and secondly to avoid problems caused by the fact that MiniZinc supports overloading but FlatZinc does not.

9.3 Reified solver-specific globals

An new feature of `mzn2fzn` version 1.0 is that if a predicate *p_reif(..., var bool: b)* is defined then it wil be used wherever an application *p(...)* appears in a reified context. This is useful when the reified form a global is not provided by a solver.

As an example, consider a solver that provides a built-in version of `all_different` for integers:

```
predicate all_different_int(array[int] of var int: x);

predicate all_different_int_reif(array[int] of var int: x, var bool: r) =
    r <-> forall(i,j in index_set(x) where i < j) ( x[i] != x[j] );
```

The alternative version will be used if `all_different_int` occurs in a reified context.

9.4 Redefinition of Flatzinc built-in constraints

`mzn2fzn` now allows FlatZinc built-in constraints to be redefined in terms of other FlatZinc built-in constraints. For example, the following declarations in MiniZinc

```
predicate int_lt(int: x, int: y);
predicate int_gt(int: x, int: y) = int_lt(y, x);
```

will cause `mzn2fzn` to replace all instances of the builtin-in constraint `int_gt` in the generated FlatZinc with `int_lt` (with the arguments inverted).

This ability allows FlatZinc implementations to minimise the number of built-in constraints that they have to support directly.

Note that you need to provide a bodyless predicate definition for any *actual* FlatZinc built-in constraints that are used within redefinitions. (We hope to remove this requirement in a later version.)

The MiniZinc standard library, `stdlib.mzn`, which is implicity included in every MiniZinc model includes a file named `redefinitions.mzn`. The default version of `redefinitions.mzn` is empty, however FlatZinc implementations that choose to implement some built-ins via redefintion should provide an alternative version of `redefinitions.mzn` along with any specialised global constraints.

10 Comments, suggestions, and bug reports

Bugs can be reported via the G12 bug tracking system at `bugs.g12.csse.unimelb.edu.au`.

Comments, questions and suggestions should be sent to the G12 Users mailing list. You can subscribe to the list by sending an e-mail containing the word `subscribe` in the body to `g12-users-request@csse.unimelb.edu.au`. Thereafter, mail may be sent to `g12-users@csse.unimelb.edu.au`.