GRACeFUL CPL-backend
User Manual

Table of contents

1. Description

This module links functional programming in Haskell with constraint programming in MiniZinc. An abstract syntax tree (“AST” now on) of the MiniZinc language makes it possible to define a constraint satisfaction problem in Haskell. MiniZinc or choco3 takes over to solve the model and a parser parses the solution(s) back into Haskell values. In detail, the generated MiniZinc code first gets translated to FlatZinc code and then solved. The output of a solution follows the conventions of the G12/FD solver. The parser of this module is based on these conventions as well. Although the output item is supported by the Haskell AST, it is strongly recommended to be used only for testing and debugging purposes.

A pretty-printer is provided, which prints the MiniZinc translation of the model. In addition., the user can choose to return one or all solutions. This module works with MiniZinc and, apart from MiniZinc's built-in solvers, it can use choco3 as well.

Example models are included and can be printed with the `printModel` function and run with the `testModel` function. Read §5 or the comments in `MZinHaskell.hs` for more details.

Not yet supported

- Annotations
- Solving set constraint problems with the choco3 solver
- Returning first *n* solutions

2. Installation

Requirements

This module depends on the software indicated in the list below:

- GHC 7.10.3
- MiniZinc 2.0
- JDK 8+ *(only if choco3 will be used)*

Installation steps

Download

Download the source code available on BitBucket:
https://bitbucket.org/graceful_team/graceful_repos

Configure

1. You will need to provide the installation directory of MiniZinc. Depending on your OS, follow the instructions in `CPL-backend/WindowsAux.hs` or in `CPL-backend/LinuxAux.hs`.
2. You will also need to import module `WindowsAux` or `LinuxAux` depending on your OS. Make the appropriate changes in `MZinHaskell.hs`, as indicated in the comments.

3. Structure: Defining a MiniZinc model

A MiniZinc model consists of multiple items. In this section we describe the Haskell representation of each kind of items that MiniZinc syntactically consists of. The representation of a MiniZinc model in Haskell is just a list of items. In Haskell words, `type MZModel = [Item]`. The code below shows the Haskell representation for each MiniZinc item.

Syntax:

```
data Item = Comment String
          | Include Filename
          | Declare Inst VarType Ident (Maybe Expr)
          | Assign Ident Expr
          | Constraint Expr
          | Solve Solve
          | Output Expr
          | Pred Ident [Param] (Maybe Expr)
          | Test Ident [Param] (Maybe Expr)
          | Function TypeInst Ident [Param] (Maybe Expr)
          | Annotation
          | Empty
```

Declaring a variable

The following item constructor is used to declare a variable in the AST of MiniZinc.

Syntax:

```
Declare TypeInst Ident (Maybe Expr)
```

A Haskell string for the name of the MiniZinc variable substitutes `Ident`. The last argument of the `Declare` constructor is a Haskell optional, since MiniZinc provides the choice to initialize a variable on declaration or later.

Type-Insts

As in MiniZinc's terminology, a type-inst of a variable is its inst and its type. In our AST we define a type-inst as a pair of an inst and a type.

Syntax:

```
type TypeInst = (Inst, VarType)
```

Insts

All variable declarations in the AST must have an explicit inst. An `Inst` can either be `Par` (translates to "par") or `Dec` (translates to "var").

Syntax:

```
data Inst = Par | Dec
```

Variable types

The AST supports all types that MiniZinc supports.

Syntax:

```
data VarType = Bool
             | Int
             | Float
             | String
             | Set VarType
             | Array [VarType] TypeInst
             | List TypeInst
```

```
                | Opt VarType
                | Range Expr Expr
                | Elems [Expr]
                | AOS Ident
                | Any
```

Built-in scalar types

`Bool`, `Int`, `Float` and `String` correspond to the built-in scalar types of MiniZinc.

Built-in compound types

`Set VarType` is used to declare a set of values. The argument of the Set constructor refers to the type of the elements of the set.

Example:

Haskell AST:	`Set Int`
MiniZinc:	set of int

`Array [VarType] TypeInst` is used to declare an array. Its first argument represents the indexes of the array. Multidimensional arrays are supported, where each element of the list corresponds to a dimension of the array. The second argument of the `Array` constructor corresponds to the type-inst of the array's elements. For an "array[int] of ..." one can use the `List TypeInst` constructor, as a list in MiniZinc is an abbreviation for an int-indexed array.

With `Opt VarType` one can declare an optional type. The syntax is similar to that of the `Set` constructor.

Constrained types

The `Range Expr Expr` constructor defines an integer range from the expression of the first argument to that of the second argument.

Example:

Haskell AST:	`Range (IConst 1) (IConst 3)`
MiniZinc:	`1..3`

Use `Elems [Expr]` to restrict the domain of a variable to the set of the values in the list `[Expr]`.

One can also restrict the domain of the declared variable to a set parameter by using `AOS Ident`. The name of the set parameter replaces `Ident`.

Assigning to a variable

Syntax:

```
Assign Ident Expr
```

A string with the name of the variable goes in place of `Ident` and an expression goes in place of `Expr`. The syntax of an `Expr` is explained in section 4.

Example:

Haskell AST: `Assign “myvar” (BConst True)`

MiniZinc: `Myvar = true;`

Setting a constraint

Syntax:

```
Constraint Expr
```

where an expression in the form of the AST replaces `Expr`.

Example:

Haskell AST: `Constraint (Bi Neq (Var "y") (Var "v"))`

MiniZinc: `constraint y != v;`

The solve item

The syntax below is used to define a solve item in the MiniZinc model.

Syntax:

```
Solve Solve
```

The naming convention here might be confusing. The first `Solve` is the item constructor and must stay as is when defining a constraint, while the second indicates that the argument of the `Solve` constructor must be of (Haskell) type `Solve`. Here are the values that type `Solve` can take.

Syntax:

```
data Solve = Satisfy
           | Minimize Expr
           | Maximize Expr
```

`Solve Satisfy` determines a constraint satisfaction problem, while the rest two cases determine an optimization problem where the object function replaces `Expr`.

User defined operations

Predicates and tests

The syntax for defining a predicate and tests in the Haskell AST is shown below, respectively.

Syntax:

```
Pred Ident [Param] (Maybe Expr)
```

Syntax:

```
Test Ident [Param] (Maybe Expr)
```

The first argument of the constructor corresponds to the name of the predicate/test. A list containing its arguments follows. If it is a natively supported predicate/test, then

`Nothing` goes in place of `(Maybe Expr)`, while in the case of a user-defined predicate/test a body must be provided.

An argument for the predicate is specified by a variable name, its inst and its type.

Syntax:

```
type Param = (Inst, VarType, Ident)
```

Functions

To define a function in the Haskell AST, one more piece of information is needed: the type-inst of the function's result.

Syntax:

```
Function TypeInst Ident [Param] (Maybe Expr)
```

Output item

Syntax:

```
Output Expr
```

Importing a file

Syntax:

```
Include Filename
```

where the path to the file replaces `Filename`.

Readability and documentation

Two more items are provided for readability and documentation purposes. Item constructor `Empty` translates to an empty line.

Comments can be added as follows

Syntax:

```
Comment String
```

4. Expr for expressions

In this section we explain how MiniZinc expressions can be represented in our Haskell AST. The Haskell datatype for expressions is `Expr`.

Built-in scalar values

Syntax:

```
data Expr = AnonVar
          | Var Ident
          | BConst Bool
          | IConst Int
          | FConst Float
          | SConst String
```

As indicated by the syntax shown above, constructor `BConst` followed by a Haskell boolean value represents the corresponding boolean value in the MiniZinc language. Similarly, `IConst`, `FConst` and `SConst` are the constructors for integer, floating and string values in MiniZinc, respectively.

Representation of a variable is done with the `Var` constructor followed by the name of the variable in a Haskell string format. Care is recommended when using `Var Ident`, since MiniZinc variable names are represented by Haskell strings, so misspelling a name will cause a FlatZinc compilation error or unexpected behaviour.

The MiniZinc anonymous decision variable `_` is represented by `AnonVar`.

Built-in compound values

Arrays

Four distinct constructors are provided for array representations.

Syntax:

```
data Expr = ...
          | Interval Expr Expr
          | ArrayLit [Expr]
          | ArrayLit2D [[Expr]]
```

```
          | ArrayComp Expr CompTail
          | ArrayElem Ident [Expr]
```

The `Interval` represents arrays expressed in MiniZinc with the `..` operator.

The `ArrayLit` constructor is used for representing array literals. The first argument of this constructor contains the representation of the array's elements. To define a 2-dimensional array literal, use `ArrayLit2D` constructor.

Array comprehensions can be represented with the `ArrayComp` constructor. The first argument of the constructor represents the head expression of the comprehension. The `CompTail` datatype represents the generators of the comprehension and gives an optional expression for a *where* restriction on the generators.

Syntax:

```
type CompTail = ([Generator], Maybe Expr)
```

A `Generator` has the following syntax.

Syntax:

```
type Generator = ([Ident], Expr)
```

Representation of an element of an array can be done with the `ArrayElem` constructor. Its first argument should be substituted by a Haskell string with the name of the array and the second argument is a list with the index(es) of the specific element of the array. This list must contain as many elements as the array's dimensions.

Sets

Sets can be represented in two ways by our Haskell AST.

Syntax:

```
data Expr = ...
          | Interval Expr Expr
          | SetLit [Expr]
          | SetComp Expr CompTail
```

Similarly to `ArrayLit`, the `SetLit` constructor is used for representing set literals.

`SetComp` is used for set comprehensions. The first argument of the constructor represents the head expression of the comprehension. The `CompTail` datatype works the same as with the `ArrayComp` constructor.

Example:

Haskell AST:
```
SetComp (Bi BPlus (Var “i”) (Var “j”))
        ([([“i”], Interval (IConst 1) (IConst 3)),
          ([“j”], Interval (IConst 1) (Var “i”))
         ], Nothing)
```

MiniZinc:
```
{i+j | i in 1..3, j in 1..i }
```

Example:

Haskell AST:
```
SetComp (Var “i”)
        ([([“i”], Interval (IConst 1) (IConst 10))],
          (Bi Eq (Bi Mod (Var “i”) (IConst 2))
                 (IConst 0)))
```

MiniZinc:
```
{ i | i in 1..10 where (i mod 2 = 0) }
```

Operators

To represent operations over values, the following syntax is used.

Syntax:

```
data Expr = ...
          | Bi Bop Expr Expr
          | U Uop Expr
```

Constructor `U` is for unary operators and `Bi` for binary operators. The first argument of both constructors represent the operator. A list with all operators' representation is included in the appendix of this document. The `Expr` arguments represent the expressions on which the operator applies.

Call expressions

To call a function or a predicate in MiniZinc, the following syntax is used in the Haskell AST.

Syntax:

```
data Expr = ...
          | Call Func [Expr]
```

A call can refer to either a user-defined or a built-in function or predicate. All built-in MiniZinc 2.0 calls are supported and named as in MiniZinc. For a user-defined call, use the function `userD :: Ident -> Func` providing a Haskell string with the name of the desired function or predicate.

Example:

Haskell AST:

```
Call forall
[ArrayComp (Bi Neq (ArrayElem "a" [Var "i"])
                   (ArrayElem "a" [Var "j"]))
           ([(["i", "j"], Interval (IConst 1) (IConst
3))], Nothing)]
```

MiniZinc:

```
forall([a[i] != a[j] | i, j in 1..3])
```

Generator calls

Syntax:

```
data Expr = ...
          | GenCall Func CompTail Expr
```

Example:

Haskell AST:

```
GenCall forall
  ([(["i", "j"], Interval (IConst 1) (IConst 3))],
Nothing)
  (Bi Neq (ArrayElem "a" [Var "i"]) (ArrayElem "a" [Var
"j"]))
```

MiniZinc:

```
forall(i, j in 1..3)
  (a[i] != a[j])
```

Conditionals

For representing an *if-then-else* conditional in MiniZinc, use the following syntax

Syntax:

```
data Expr = ...
          | ITE [(Expr, Expr)] Expr
```

The list in the first argument of `ITE` constructor must have at least one pair of expressions. The first term of the pair represents the conditional (*if*) and the second term represents the expression in case of satisfaction (*then*). The last argument of the `ITE` constructor represent the expression in case of non-satisfaction (*else*). If the list of the first argument has more than one elements, all next elements after the first translate to consecutive *elseif-then* MiniZinc expressions.

Example:

Haskell AST:

```
ITE [(Bi Lt (Var "x") (IConst 0), U UMinus (IConst 1)),
     (Bi Gt (Var "x") (IConst 0), IConst 1)]
    (IConst 0)
```

MiniZinc:

```
if x < 0 then - 1
elseif x > 0 then 1
else 0 endif
```

Let expressions

Syntax:

```
data Expr = ...
          | Let [Item] Expr
```

The `Item`s in the list of `Let`'s first argument must be only variable declaration or constraint items. The last argument of the `Let` constructor corresponds to the expression following the "in" keyword in MiniZinc's let expression.

Example:

Haskell AST:

```
Let [Declare (Dec, Int) "x" (Just (IConst 3)),
     Declare (Dec, Int) "y" (Just (IConst 4))]
    (Bi BPlus (Var "x") (Var "y"))
```

MiniZinc:

```
let {var int: x = 3;
     var int: y = 4;}
in x + y
```

5. Representing and running a model

In your Haskell source code, import module `MZinHaskell`. Use the Haskell AST described above to define your MiniZinc model. In case you need a separate data file for your model, create a Haskell list with `Assign` items. Use

```
> writeData <assign-model>
```

to write the data to a .dzn file. This function takes a list of `Item`s (or a `MZModel`) as an argument. You will be asked for the path of the file in which the data will be written. Use the corresponding conventions for representing a filepath, depending on your OS. For example, in Windows the filepath will be similar to `C:\file\path\to\datafile.dzn`.

To solve the model interactively, use

```
> iTestModel <model>
```

This function will initiate a dialogue where, first, you will be asked for the path of the file in which the generated MiniZinc code should be written. Provide this filepath without the ".mzn" extension. In case the file extension is included in the filepath, the name of the target file will just have a ".mzn.mzn" suffix. After entering the filepath for the .mzn file, you will be prompted to provide the path to the data file. Enter empty input, in case no data file is needed, or provide the asked filepath (file extension included). Next, you will be given the choice to use the built-in FD solver or choco3. For FD, type fd. Enter empty input for choco3. Last, you can choose to output only one solution (the best, in case of an optimization problem) by giving empty input, or all solutions by entering 0 at the stdin.

An example of running `iTestModel` is shown below. The text in bold shows the dialogue generated by this function.

```
> :l test.hs
> iTestModel planning
Enter MiniZinc model's filepath (without .mzn extention):
C:\Users\klara\Documents\MiniZinc\plan
Is there a data file? If yes, provide its filepath:
C:\Users\klara\Documents\MiniZinc\pland.dzn
Type "fd" for G12/FD solver or leave empty for choco solver.
fd
Enter 0 to output all solutions.
```

File `test.hs` contains a few example models.

Appendix

Unary operators

Haskell AST	MiniZinc
Not	not
UPlus	+
UMinus	-

Binary operators

Haskell AST	MiniZinc	Haskell AST	MiniZinc
Gt	>	BPlus	+
Lt	<	BMinus	-
Lte	<=	Times	*
Gte	>=	Div	/
Eq	=	IDiv	div
Eqq	==	Mod	mod
Neq	!=	LRarrow	<->
Larrow	<-	Rarrow	->
And	/\	Or	\/
In	in	Sub	subset
Super	superset	Union	union
Inter	intersect	Diff	diff
SDiff	symdiff	RangeOp	..
Concat	++		

Calls

Haskell AST	MiniZinc	Haskell AST	MiniZinc
BoolToInt	bool2int	Sum	sum
IntToFloat	int2float	Product	product
SetToArray	set2array	Min	min
Forall	forall	Max	max
Xorall	xorall	Abs	abs
Assert	assert	Sqrt	sqrt
Abort	abort	Power	pow
Trace	trace	Exp	exp
Fix	fix	Ln	ln
IsFixed	is_fixed	Log	log
Show	show	Sin	sin
ShowInt	show_int	Cos	cos
ShowFloat	show_float	Tan	tan
Exists	exists	Asin	asin
Acos	acos	Atan	atan
Sinh	sinh	Cosh	cosh
Tanh	tanh	Asinh	asinh
Acosh	acosh	Atanh	atanh
Card	card	ArrUnion	array_union
ArrInters	array_intersect	Length	length
Index	index_set	Index12	index_set_1of2

Index22	index_set_2of2	Deopt	deopt
ArrDom	dom_array	SizeDom	dom_size
Ceil	ceil	Floor	floor
Round	round	MConcat	concat
Join	join	Lb	lb
Ub	ub	LbArray	lb_array
UbArray	ub_array	Occurs	occurs
Absent	absent		