haskelzinc user guide

The haskelzinc package provides a library for creating Haskell representations of MiniZinc constraint models. This document describes how each MiniZinc component can be represented.

The use of the `OverloadedStrings` option is recommended for reasons explained in section Referring to a variable.

Types and Type-insts

Instantiations

All variables must have an instantiation (Inst). This is either a parameter, declared in MiniZinc/haskelzinc with *par*, or a decision variable, declared in MiniZinc/haskelzinc with *var*.

Example: declaring a variable x as parameter (instantiation is par)

haskelzinc:
```
par Int "x" =. 6
```

MiniZinc:
```
par int: x = 6;
```

Example: declaring a variable x as decision variable (instantiation is var)

haskelzinc:
```
var Int "x" =. 6
```

MiniZinc:
```
var int: x = 6;
```

Variables can be declared and then assigned later.

Example: declaring and assigning it later

haskelzinc:

```
[ par Int "x"
, x  =. 6 ]
```

MiniZinc:

```
par int: x;
x = 6;
```

Type-inst Expressions Overview

This is an overview of the haskelzinc supported types.

Syntax:

```
data Type = Bool
          | Int
          | Float
          | String
          | Set Type
          | Array [Type] Inst Type
          | List Inst Type
          | Ann
          | CT Expr
          | Range Expr Expr
          | VarType String
```

Built-in Scalar Types and Type-insts

Built-in scalar types

`Bool`, `Int`, `Float` and `String` correspond to the built-in scalar types of MiniZinc. Enumerated types are not supported yet.

Built-in compound types

Sets

`Set Type` 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: declaring a set

haskelzinc:
```
par (Set Int) "x" =. 1 ... 9
```

MiniZinc:
```
par set of int: x = 1 .. 9;
```

Arrays

`Array [Type] Inst Type` 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 Inst Type` constructor, as a list in MiniZinc is an abbreviation for an int-indexed array.

Example: declaring an array

haskelzinc:
```
par (Array [CT (0...2)] Par Int) "a1" =. array [1,2,3]
```

MiniZinc:
```
array[0 .. 2] of par int: a1 = [1, 2, 3];
```

Option Types

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

Example: creating an optional int type

haskelzinc:
```
Opt Int
```

MiniZinc:
```
opt int
```

The Annotation Type

The `Ann` constructor defines the annotation type *ann*.

Constrained Type-insts

Range

NOTE: The `CT (x ... y)` syntax is preferred over the `Range` equivalent.

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

Example: creating range 1 to 3

haskelzinc:
```
par (Array[Range 0 2] Par Int) "a1" =. array [1,2,3]
```

MiniZinc:
```
array[0 .. 2] of par int: a1 = [1, 2, 3];
```

CT

With `CT Expr` we can express a constrained type from an expression.

Example: using CT

haskelzinc:	`par (Array [CT (intSet [1,2,3])] Par Int) "a1" =. array [4,5,6]`
MiniZinc:	`array[{1,2,3}] of par int: a1 = [4, 5, 6];`

Expressions

The haskelzinc type `Expr` is used for representing MiniZinc expressions.

Referring to a variable

With the `OverloadedStrings` language option, one can refer to representations of MiniZinc variables only by providing their name as a Haskell `String`.

Example:

haskelzinc:	`constraint $ "x" >. 2`
MiniZinc:	`constraint x > 2;`

The anonymous identifier / wildcard of MiniZinc is represented with `__` (two underscores).

Literals of built-in scalar types

The two boolean values are similarly represented in haskelzinc. Write `true` for MiniZinc's boolean literal "true" and `false` for the "false" literal. Haskell integer and floating point literals are sufficient for their MiniZinc representation. For the representation

of MiniZinc string literals, use haskelzinc function
`string :: String -> Expr`.

Example:

haskelzinc:	`string "alphanumeric"`
MiniZinc:	`"alphanumeric"`

Set literals and comprehensions

Depending on the type of the set's elements, one can use one of the haskelzinc functions below to represent a MiniZinc set. They all take a list of values (of the appropriate type), which essentially represent the set's elements.

```
intSet :: [Int] -> Expr
floatSet :: [Float] -> Expr
stringSet :: [String] -> Expr
```

Example: Set literal

haskelzinc:	`intSet [1, 3, 5]`
MiniZinc:	`{1, 3, 5}`

haskelzinc also provides a more generic function for creating and representing MiniZinc set literals: `mapSet :: (a -> Expr) -> [a] -> Expr`

Function `set :: [Expr] -> Expr` can be passed an arbitrary list of expressions to represent a MiniZinc set. haskelzinc does not check for MiniZinc type errors. The list of expressions given to the `set` function should represent MiniZinc values of the same type.

The operator `(#/.) :: Expr -> [CompTail] -> Expr` can be used for set comprehensions. Operator `(@@) :: [Ident] -> Expr -> CompTail` creates a generator expression where the identifiers in the list of left argument range within the (set) expression of the right argument.

Example:

haskelzinc:
```
2 *. "i" #/. [["i"] @@  0 ... 5]
```

MiniZinc:
```
{2 * i | i in 0 .. 5}
```

A comprehension tail may have multiple generator expressions, thus in haskelzinc a comprehension tail is represented with a list of generator expressions. To restrain a generator expression with MiniZinc's "where" clause, use function `where_ :: CompTail -> Expr -> CompTail` provided by haskelzinc. This function can be used with single quotes for infix notation, as shown in the example below.

Example:

haskelzinc:
```
"i" *. "j" #/. [["i", "j"] @@ 0 ... 5
                `where_` ("i" !=. "j")]
```

MiniZinc:
```
{i * j | i, j in 0 .. 5 where i != j}
```

Array literals and comprehensions

Similarly to set literals, haskelzinc provides a number of functions to represent simple array literals, depending on the type of the array's elements.

For 1-dimensional arrays, use
`boolArray :: [Bool] -> Expr`
`intArray :: [Int] -> Expr`
`floatArray :: [Float] -> Expr`
`stringArray :: [String] -> Expr`

Function `mapArray :: (a -> Expr) -> [a] -> Expr` is a more generic function for creating simple array literals. The function `array :: [Expr] -> Expr` can be passed an arbitrary list of expressions to represent a MiniZinc 1-dimensional array. haskelzinc does not check for MiniZinc type errors. The list of expressions given to the `array` function should represent MiniZinc values of the appropriate type.

The 1-dimensional array functions above have their 2-dimensional counterparts:
`boolArray2 :: [[Bool]] -> Expr`
`intArray2 :: [[Int]] -> Expr`
`floatArray2 :: [[Float]] -> Expr`
`stringArray2 :: [[String]] -> Expr`
`mapArray2 :: (a -> Expr) -> [[a]] -> Expr`
`array2 :: [[Expr]] -> Expr`

For array comprehensions, use operator `(#/.) :: Expr -> [CompTail] -> Expr`. An array's comprehension tail can be represented in haskelzinc with the same operators and functions (`(@@)` and `where_`) as in the case of set comprehensions.

Array access

To represent an array's element, use operator `(!.) :: String -> [Expr] -> Expr`. The length of the list in the right argument should match the represented list's dimensions.

Example:

haskelzinc:
```
"matrix"!.["i", "j"]
```

MiniZinc:
```
matrix[i,j]
```

Generator calls

In MiniZinc, generator calls are a convenient way to call MiniZinc functions that expect an array as an argument. To represent a generator call in haskelzinc, use function `forall :: [CompTail] -> String -> Expr -> Expr`.

Example:

haskelzinc:
```
forall [["i"] @@ "S1", ["j"] @@ "S2"] "sum"
     ("x"!.["i", "j"])
```

MiniZinc:
```
sum(i in S1, j in S2) (x[i, j])
```

Conditional

For MiniZinc's if-then(-elseif)-else expression, the following haskelzinc functions are provided:

```
if_ :: Expr -> (Expr -> [(Expr, Expr)])
then_ :: (Expr -> [(Expr, Expr)]) -> Expr -> [(Expr, Expr)]
elseif_ :: [(Expr, Expr)] -> Expr -> (Expr -> [(Expr, Expr)])
else_ :: [(Expr, Expr)] -> Expr -> Expr
```

Use the last three functions in single quotes, for a conventional syntax of the conditional.

Example:

haskelzinc:
```
if_ true `then_` 1 `else_` 0
```

MiniZinc:
```
if true then 1 else 0 endif;
```

Note that haskelzinc does not use "endif".

Let expressions

For MiniZinc let-expressions, use haskelzinc function `let_ :: [GItem i] -> Expr -> Expr`. The items provided in the list should be variable declarations and/or constraint items. haskelzinc does not check for syntactical correctness of the let-expression.

Example:

haskelzinc:
```
predicate "posProd"[var Int "x", var Int "y"] =.
let_ [ var Int "z"
     , constraint $ "z" =.= "x" *. "y"]
     ("z" >. 0)
```

MiniZinc:
```
predicate posProd(var int: x, var int: y)
  = let {var int: z;
         constraint z = x * y;}
    in z > 0;
```

Call expressions

Module `Interfaces.MZBuitIns` contains representations of MiniZinc's built-in operators, functions, predicates, tests and annotations. For calling a user-defined operation, use the functions below:

To represent a prefix call to a function, test or predicate:
`prefCall :: String -> [Expr] -> Expr`

To represent an infix (quoted) call to a function, test or predicate:
`infCall :: String -> Expr -> Expr -> Expr`

To represent a prefix (quoted) call of an operator:
`prefOp :: String -> Op`

To represent an infix call to an operator.
`infOp :: String -> Op`

In all cases, the first argument is the MiniZinc name/identifier or symbol of the called operation. In `prefCall`, the second argument represents the arguments with which the operation is called. The same hold for the two `Expr` arguments of `infCall`.

A previous example shows how to represent the declaration of the user-defined predicate "posProd". Assuming a model that contains this definition, one can call the predicate as shown in the example below.

Example: Including a file

haskelzinc:
```
posProd :: [Expr] -> Expr
posProd = prefCall "posProd"
⋮
x =. posProd[1, 2]
```

MiniZinc:
```
x = posProd(1, 2);
```

Items

A MiniZinc model in haskelzinc is a list of assigned GItems, or `[GItem 'OK]`.

Include Items

Use `include :: String -> GItem 'OK` to include a file.

Example: Including a file

haskelzinc:
```
include "foo.zinc"
```

MiniZinc:
```
include “foo.zinc”;
```

Variable Declaration Items

The functions `var :: Varr i -> Type -> String -> GItem i` and `par :: Varr i -> Type -> String -> GItem i` create a variable declaration.

See *Instantiations* in *Types and Type-insts* for example usage.

Enum Items

Enumerated types are not supported.

Assignment Items

The function `(=.) :: Assignable a => a -> Expr -> GItem 'OK` represents an assignment.

See *Instantiations* in *Types and Type-insts* for example usage.

Constraint Items

The function `constraint :: Expr -> GItem 'OK` creates a constraint.

Example: Creating a constraint

haskelzinc:
```
constraint ("a" *. "x" <. "b")
```

MiniZinc:
```
constraint a * x < b;
```

Solve Items

The function `solve :: Solve -> GItem 'OK` creates a solve item.

It can be combined with `satisfy :: Solve`,

Example: Create satisfaction problem solve item

haskelzinc:
```
solve satisfy
```

MiniZinc:
```
solve satisfy;
```

with `maximize :: Expr -> Solve`,

Example: Create satisfaction problem solve item

haskelzinc:
```
solve (maximize ("a" *. "x"))
```

MiniZinc:
```
solve maximize a * x;
```

or with `minimize :: Expr -> Solve`,

Example: Create satisfaction problem solve item

haskelzinc:
```
solve (minimize ("a" *. "x"))
```

MiniZinc:
```
solve minimize a * x;
```

Output Items

The function `output :: [Expr] -> GItem 'OK` creates an output item.

Example: Creating an output item

haskelzinc:
```
output [string "The value of x is ", mz_show ["x"], string "!\n"]
```

MiniZinc:
```
output ["The value of x is ", show(x), "!\n"];
```

Annotation Items

Use function `annotation :: String -> [GItem 'DS] -> GItem 'OK` to declare a solver specific annotation.

User-defined Operations

The function `predicate :: String -> [GItem 'DS] -> GItem 'DS` creates a predicate. Since it creates an unassigned `GItem 'DS` we use the `(=.)` operator to declare the body of the predicate.

Example: Creating a predicate

haskelzinc:
```
predicate "even" [var Int "x"] =. "x" `_mod_` 2 =.= 0
```

MiniZinc:
```
predicate even(var int: x) = x mod 2 = 0;
```

Similarly to a predicate, the following haskelzinc functions can be used for the declaration of MiniZinc functions and tests.

```
function :: Inst -> Type -> String -> [GItem 'DS] -> GItem 'DS
```

```
test :: String -> [GItem 'DS] -> GItem 'DS
```

Annotations

The operator `(|:) :: Annotatable a => a -> Annotation -> a` adds an annotation to an expression, a solve item, a user-defined operation or a variable declaration.

The arguments of the annotation call should be prepended with `E` if they are an expression or `A` if they are an annotation.

Example: Creating an annotation on a solve item

haskelzinc:
```
solve (satisfy |: mz_int_search
  [ E "x"
  , A (mz_first_fail [])
  , A (mz_indomain_min [])
  , A (mz_complete [])
  ])
```

MiniZinc:
```
solve :: int_search(x, first_fail, indomain_min,
complete) satisfy;
```

Time Space Constraints

Time Space Constraints

Time space constraints are used to model situations where a series of actions have to be performed at several different locations, under certain restrictions. For instance, *<1, 1, 3, $, 1, $, $, 2, 1, 2, nop, nop>* is a possible sequence. The numbers *1*, *2* and *3* correspond to three different types of actions. The *$* operator marks a switch of location and the *nop* operator is used to complete the suffix of the series to a fixed size. At the first station, actions *1*, *1* and *3* are performed, in that order. Action *1* is performed at the second station. Station three performs no actions and the fourth station executes actions *2*, *1* and *2*, in that order.

`actionSequence Int String ASExpr` is used to construct a time space constraint. The first argument corresponds to the number of different actions. The second argument is the name of the sequence variable. The final argument is the actual constraint, represented by the haskelzinc type `ASExpr`.

Syntax:

```
data ASExpr = AtleastCells Int
            | AtmostCells Int
            | Atleast Int Int
            | Atmost Int Int
            | Incompatible Int Int
            | Implication Int Int
            | ValuePrecedence Int Int
            | StretchMin Int Int
            | StretchMax Int Int
            | Or Int Int
```

`atleast_cells Int` and `atmost_cells Int` constrain the number of locations in the sequence, where the first argument represents the minimum or maximum number of locations.

Example: Creating a time space constraint - minimum amount of locations

haskelzinc: `actionSequence 2 "x" (atleast_cells 2)`

`atleast Int Int` and `atmost Int Int` constraints the amount of times a given action is performed in each location. The first argument represents the action. The second argument is the minimum or maximum number of times this action can be performed in each location.

Example: Creating a time space constraint - maximum amount of executions

haskelzinc: `actionSequence 2 "x" (atmost 1 2)`

`incompatible Int Int` imposes the restriction that the two given actions can not be performed in the same location, where the two arguments correspond to the given actions.

Example: Creating a time space constraint - incompatible actions

haskelzinc: `actionSequence 2 "x" (incompatible 1 2)`

`implication Int Int` represents an implication constraint. The two arguments represent the two given actions, where the first implies the second. This means that if this first action is performed in a location, the second has to be performed as well.

Example: Creating a time space constraint - implied actions

haskelzinc: `actionSequence 2 "x" (implication 1 2)`

`value_precedence Int Int` imposes an order on the two given actions, represented by the two arguments. In order for the second action to be performed in a location, the first action has to be performed first.

Example: Creating a time space constraint - ordered actions

haskelzinc: `actionSequence 2 "x" (value_precedence 1 2)`

`stretch_min Int Int` and `stretch_max Int Int` restrict the amount of times a given action is performed. The first argument represents the given action. The second argument represents the amount of times the action can at least / at most be performed in a row, when the action is performed at least once.

Example: Creating a time space constraint - minimal series

haskelzinc: `actionSequence 2 "x" (stretch_min 1 3)`

`or_as Int Int` constraints the occurrence of the two given actions, represented by the two arguments. In each location, at least one of these two given actions has to be performed.

Example: Creating a time space constraint - or constraint

haskelzinc: `actionSequence 2 "x" (or_as 1 2)`

Defining a Model

The following example illustrates how to construct a simple time space sequence model. Firstly, the sequence variable `x` is declared. The sequence variable has a fixed length (in this case 20), which acts as a maximum length for the resulting action sequence, since it can be padded with nop operators. This example contains two different actions, to be divided over four locations. This means that each field in the sequence can contain either a *1* or *2* (corresponding to the two actions), a *3* (corresponding to the *$* operator, signaling a location switch) or a *4* (corresponding to the *nop* operator). Two constraints are applied to the sequence. The first demanding that at most 4 locations are used. The second restricting each location to at least execute one of the two actions.

Example: Defining a time space sequence model

haskelzinc:

```
mod = [ include “regular.mzn”
      , var (Array [CT $ 1...20] Dec (CT $ 1...4)) “x”
      , actionSequence 2 “x” (atmost_cells 4)
      , actionSequence 2 “x” (or_as 1 2)
      , solve satisfy
      ]
```

Cost Constraints

`actionSequenceCost String String ASCostExpr` is used to construct cost constraints to optimize a time space constraint model. The first argument represents the name of the sequence variable. The second argument represents the name of the resulting cost variable. The final argument is the actual cost constraint, represented by the haskelzinc type `ASCostExpr`.

Syntax:

```
data ASCostExpr = UniformCost Int Int
                | DiscountCost Int Int Int Int Bool
                | DependentCost Int Int Int Int Int Bool
```

`uniformCost Int Int` constraints the cost of the given action to be uniform throughout the sequence. The first argument represents the given action, the cost of which is always equal to the second argument.

Example: Creating a time space cost constraint - uniform cost

haskelzinc:

```
actionSequenceCost “x” “cost1” (uniformCost 1 5)
```

`discountCost Int Int Int Int Bool` constraints the cost of the given action to initially be a given cost, which reduces after the initial execution to a discounted cost. The first argument represents the action for which the cost is being constraint. The second argument corresponds to the *$* operator, which switches locations in the action sequence

(the number of actions + 1). The third and fourth arguments represent the original and discounted cost (which applies after the first execution) respectively. The final argument denotes whether the action starts with the initial cost in each location or only once in the entire sequence.

Example: Creating a time space cost constraint - discount cost

haskelzinc:
```
actionSequenceCost “x” “cost2”
                   (discountCost 2 3 5 4 True)
```

`dependentCost Int Int Int Int Int Bool` constraints the cost of the given action to be either the full or the discounted cost, depending on whether the influencing action is executed. The first argument represents the action for which the cost is being constraint. The second argument is the influencing action, which determines the cost of the first. The third argument corresponds to the *$* operator, which switches locations in the action sequence (the number of actions + 1). The third and fourth arguments represent the full and discounted cost respectively. The final argument denotes whether the influencing effect is local or global. If the flag is true, the cost of the given action is constraint to be the discounted cost if the influencing action is executed in the same location (at any time) and the full cost otherwise. If the flag is false, the discounted cost is used if the influencing action occurs anywhere in the entire sequence.

Example: Creating a time space cost constraint - dependent cost

haskelzinc:
```
actionSequenceCost “x” “cost2”
                   (dependentCost 2 1 3 6 4 False)
```

Cost Constraints Import

The functionality for using cost constraints is not by default available in haskelzinc and has to be manually imported. Imports are performed by using the `useCostPreds [ASCostPredExpr]` function. It takes a list of dependencies as an argument, represented by the haskelzinc type `ASCostPredExpr`, which correspond directly to the three constraints defined above.

Syntax:

```
data ASCostPredExpr = UniformCostPred
                    | DiscountCostPred
                    | DependentCostPred
```

Defining a Model

The following example illustrates how to extend the previous example with a cost calculation. Firstly, the required cost constraints need to be imported. Secondly, two variables `c1` and `c2` are defined, corresponding to the costs for the first and second action respectively. The first action is assigned a uniform cost: every usage of action *1* results in a cost of 5. The second action is assigned a dependent cost: a usage of action *2* costs 5 if action *1* is executed in the same location and 10 otherwise. Finally, an additional variable `cost` is defined to represent the total cost of the entire sequence.

Example: Defining a time space sequence model with cost

haskelzinc:

```
mod = (useCostPreds
          [uniformCostPred, dependentCostPred])
   ++ [ include “regular.mzn”
      , var (Array [CT $ 1...20] Dec (CT $ 1...4)) “x”
      , actionSequence 2 “x” (atmost_cells 4)
      , actionSequence 2 “x” (or_as 1 2)
      , var Int “c1”, var Int “c2”
      , actionSequenceCost “x” “c1” (uniformCost 1 5)
      , actionSequenceCost “x” “c2”
          (dependentCostPred 2 1 3 10 5 True)
      , var Int “cost” =. “c1” +. “c2”
      , solve satisfy
      ]
```