Grooml – Groovy Mathematical Modelling Language

Table of contents

1 Introduction

The central class for solving Grooml-models is GroomlInterpreter. It uses some dynamic object orientation features of Groovy (similar to what builders in Groovy do, see http://groovy.codehaus.org/Builders) in order to provide some special commands, operators etc. for conveniently defining mathematical models.

In the sequel, we assume that the given code examples are evaluated in the context of a GroomlInterpreter. The release of Combean/Grooml contains a sample script ManualExamples.groovy that executes all examples shown below. We use a utility function check(...) for displaying and verifying the results of the sample statements. The exact definition of check(...) can be seen in ManualExamples.groovy but the syntax of it should be intuitively clear without further explanation.

2 Elements of LP/MIP-models

This section describes the elements that LP- or MIP-models are composed of.

2.1 Sets

Sets contain elements (values). The following types can be converted to sets:

- an object of the type List (typically List literals or Ranges are used for this) or
- a closure which returns an object that can be converted to a set or

- a variable binding (see below, all variable bindings are also sets).

The interpreter provides the command `set`, which converts the argument to a set. Typically, this is not needed because this conversion is triggered automatically by the context where the definition of the set occurs.

```
// +++ Example: simple set definitions +++

s = set(1..3) // Range
check ([1, 2, 3], s)
s = set(["a", "b", "c"]) // List with elements of type String
check(["a", "b", "c"], s)
s = set("a", "b", "c") // shorter syntax
check(["a", "b", "c"], s)
s = set([i:2..4]) // variable binding
check ([2, 3, 4], s)
s = set(i:2..4) // shorter syntax
check ([2, 3, 4], s)
```

Elements defined by a closure are dynamic, i.e., the closure is only evaluated when the content of the set is accessed. The closure may thus refer to values or variable bindings that exist when the set is used but do not yet exist when the set if defined. The closure must return an object that can be converted to a set.

```
// +++ Example: dynamic set definition with unbound variable +++

s = set { 1..i }
intp."i" = 10
check([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], s)
```

Sets support the multiplication operator '*', which returns the product set of the given sets.

```
// +++ Example: construct product sets with operator '*' +++

s = (1..2) * ["a", "b"]
check([[1, "a"], [1, "b"], [2, "a"], [2, "b"]], s)
s2 = ["x"] * s
check([["x", 1, "a"], ["x", 1, "b"], ["x", 2, "a"], ["x", 2, "b"]], s2)
s = ["x"] * (1..2)
check([["x", 1], ["x", 2]], s)
s = ["x"] * (1..2) * ["a", "b"]
check([["x", 1, "a"], ["x", 1, "b"], ["x", 2, "a"], ["x", 2, "b"]], s)
```

The set-command is not required here for converting the result of the product to a set because the result of the product is automatically converted. However, adding set-directives to the above example would still produce legal code.

2.2 Variable bindings

A variable binding consists of a set of values and one or more variables to which the values in the set are bound. The following types can be converted to bindings:

- Maps with index variables as keys and values that can be converted to sets (defining the value sets for the index variables).

```
// +++ Example: definition of variable bindings +++

// create variable binding
b = bind(i:2..4)
check ([2, 3, 4], b)
// create binding for two variables
b = bind(i:1..2, j:3..3)
check([[1, 3], [2, 3]], b)
// create binding to existing set
s = set(1..3)
b = bind("i", s)
check([1, 2, 3], b)
```

Typically, variable bindings are used for defining sums of expressions, multiple rows or multiple variables in an LP. They can, however, also be used for dynamic product sets where the inner set refers to the outer set in its definition:

```
// +++ Example: product of variable bindings and dynamic sets

s = [i:1..2] * { 1..i }
check([[1,1], [2,1], [2,2]], s)
```

2.3 Variables

Variables are defined by the commands `var`, `intvar`, `vars` and `intvars`. Variables consists of

- a name
- a coefficient (for the objective function)
- a range (the domain where the variable is defined)
- a flag whether the variable is an integer variable (this flag is evaluated when a MIP-solver is applied to the model)

The syntax of the var command is

var(<name>) { <closure returning the variable definition> },

where the variable definition is either

- a single value for the coefficient or
- a list of two values, where the first value is the coefficient and the second value is the range of the variable.

The syntax of the vars command is defined analogously:

vars(<name>, <definition of a variable binding>) { <closure for the variable definition }.

The closure is evaluated for every value in the binding and, of course, the closure can (and typically will) refer to the bound index variables.

The definition of the variable binding may either be

- a value that can be converted to a variable binding or
- a value which can be converted to any other type of set (in this case the values in the set are bound to the default binding 'it' resp. 'it1', 'it2', ... if the set is a product set.

```
// +++ Example: definition of variables +++
```

```
// single variable
v = var("x") { 2.0 }
check("vars 'x' : [2.0 | [0.0; Infinity]]", v.toString())
// single integer variable
v = intvar("w") { [1.0, -10..10] }
check("vars 'w' : [1.0 | [-10.0; 10.0]]", v.toString())
// set of variables
s = set(["alice", "bob"])
v = vars("y", s) { [1.0, Double.NEGATIVE_INFINITY..1] }
check("vars 'y' : [1.0 | [-Infinity; 1.0], 1.0 | [-Infinity; 1.0]]",
v.toString())
// set of integer variables
// with coefficients and range that depends on binding
v = intvars("z", [i:1..2]) { [2*i, -i..i] }
check("vars 'z' : [2.0 | [-1.0; 1.0], 4.0 | [-2.0; 2.0]]", v.toString())
// variable definition with default binding to 'it'
v = vars("u", 1..2) { -0.5*it }
check("vars 'u' : [-0.5 | [0.0; Infinity], -1.0 | [0.0; Infinity]]",
v.toString())
// variable definition with two indices
v = vars("q", (1..2)*["a", "b"]) { 1.0 }
check(4, v.numVars)
```

2.4 Expressions

Elementary expressions consist of

- a reference to a variable (that must already be defined)
- a coefficient

They are constructed by referring to the variable through their name, indexing the variable with brackets and multiplying the result with the coefficient.

```
// +++ Example: simple expressions +++
// (referring to predefined variables x, y and z)

e = x;
check("1.0 x[]", e)
e = 2*x;
check("2.0 x[]", e)
e = y["alice"];
check("1.0 y[alice]", e)
e = -y["bob"];
check("-1.0 y[bob]", e)
```

Multiple elementary expressions can be combined using the operators '+' and '-':

```
// +++ Example: sums of simple expressions +++

e = z[1] + 2*z[2];
check("1.0 z[1] + 2.0 z[2]", e)
e = y["alice"] - y["bob"];
check("1.0 y[alice] + -1.0 y[bob]", e)
```

Larger sums of elementary expressions are typically defined using the sum-operator which takes a variable binding and a dynamic definition of an expression which refers to the index variables in the binding.

The syntax of the sum-operator is

sum(<definition of the binding>) { <closure defining the inner expression of the sum> }

```
// +++ Example: sum-operator +++

// sum with separately defined binding
b = bind(i:1..2)
e = sum(b) { -i * z[i] }
check("-1.0 z[1] + -2.0 z[2]", e)
// sum with directly defined binding
e = sum(i:["alice", "bob"]) { y[i] }
check("1.0 y[alice] + 1.0 y[bob]", e)
// sum with two index variables
e = sum(i:1..2, j:1..2) { (i+j) * q[i,j] }
check("2.0 q[1,1] + 3.0 q[1,2] + 3.0 q[2,1] + 4.0 q[2,2]", e)
```

3 Loading and solving models

3.1 Load-command

The load command starts the 'load-mode' of the interpreter. The syntax is

load { <closure containing the variables and rows to be loaded> }.

When the closure is evaluated, all commands that create variables (see Section 'Variables') and all commands that define LP-rows (see below) are automatically loaded into the LP-solver.

3.2 Definition of objective

The coefficients of the variables in the objective function are given, when the variables are created. The direction of the optimization is set by the commands

min()

or

max().

3.3 Definition of LP-rows

A single row is created by the command

row(<name of the row>) { <closure that creates the row> }.

The name of the row is a String-identifier that is used in traces (and which may be used in the future when column vectors are added to the model).

Multiple rows can be created with the command

rows(<name of the rows>, <definition of a binding>) { <closure that generates the rows }

The closure is evaluated for each value in the binding. Typically, the closure will refer to the index variables that are set by the binding.

The closure has to contain code of the following format

<expression> <relation> <right-hand side>

<expression> is any expression as described above in Section 'Expressions'.

<relation> must be one of the following operators:

- << for 'less or equal to'
- >> for 'greater or equal to'
- | for 'equal to' (unfortunately, the operators '=' or '==' cannot be overloaded in that context with Groovy).

The right-hand side must be a number (or a Groovy-expression that evaluates to a number).

3.4 Configuring and running the solver

The command

```
solver(<name of the solver>)
```

sets the solver that is used for solving the model. Supported values are

- qsopt (QSOpt solver, cf. http://www2.isye.gatech.edu/~wcook/qsopt/)
- glpk (GLPK, cf. http://www.gnu.org/software/glpk/glpk.html)
- clp, cbc and symphony for the corresponding solvers of the Coin-OR-project (cf. http://www.coin-or.org/)

The command

```
solve()
```

runs the solver on the current model.

3.5 Accessing the solution

After a solution has been calculated by executing solve(), the solution can be accessed as follows.

The command

```
solutionStatus()
```

returns a code that indicates the status of the solver. This code can be one of

- undef: the model has not yet been loaded and solved
- solved: the model has been solved successfully
- infeasible: the model has been tried to solve but it is infeasible
- unbounded: the model has been tried to solve but it is unbounded
- failure: the model has been tried to solve but the solver has failed (e.g. if the underlying MIP solver gives up after it runs out of time or out of memory).

The command

```
solutionValue()
```

retrieves the value of the objective function.

The values of individual variables can be accessed by calling a function with the name of the variable, e.g. x(1), y(“foo”) or z(2,3). If no arguments are given and there are more than one variable with the given name, then the values of all variables with that name are returned as Map, with the index values as keys and and the solution values as values in the Map.

3.6 Examples

3.6.1 A simple LP model

```
// +++ Example: Solve simple model +++

intp = new GroomlInterpreter();
intp.load {
    // define the optimization objective
    max()
    // define the variables
    vars("x", 1..2) { [it, 0..1] }
    // define a single row
    row("foo") {
        sum([i:1..2]) { x[i] } << 1
    }

    // run the LP-solver
    solve()

    // access the solution
    check("solved", solutionStatus())
    check(0.0, x(1))
    check(1.0, x(2))
    check([1:0.0, 2:1.0], x())
    check(2.0, solutionValue())
}
```

3.6.2 An integer Knapsack problem

```
// +++ Example: Integer Knapsack

intp = new GroomlInterpreter();
intp.load {
    def items = ["ring", "money", "diamond", "painting", "statue"]
    def value = [ring:4, money:2, diamond:10, painting:10, statue:20]
    def weight = [ring:1, money:2, diamond:1, painting:5, statue: 20]

    max()
      // Indicator variables for the chosen items
      intvars("x", items) { [value[it], 0..1] }
      // Maximum weight to be carried away
      row("maxweight") {
          sum([i:items]) { weight[i] * x[i] } << 4
      }
      // Choose solve Coin Cbc
    check("solved", solutionStatus())
    solver("cbc")
      // Run the MIP solver
    solveMIP()

    check([ring:1.0, money:1.0, diamond:1.0, painting:0.0, statue:0.0], x())
}
```

3.6.3 An assignment problem

```
// +++ Example: Assignment problem

intp = new GroomlInterpreter();
intp.load {
    def workers = ["alice", "bob", "carol"]
    def weekdays = ["mon", "tue", "wed", "thu", "fri"]

    def preferredDays = [ alice : ["mon", "tue", "thu", "fri"],
```

```
                        bob: ["fri"],
                        carol : ["wed", "fri"],
                       ]

    def assignmentScore = { worker, weekday ->
        preferredDays[worker].contains(weekday) ? 1 : 0
    }

    max()
    // Indicator variables x[w,d]: assign day d to worker w
    vars("x", [w:workers, d:weekdays]) { assignmentScore(w, d) }
    // Slack variables for the number of tasks that one worker may get
    vars("y", [w:workers]) { [0.0, 0..1] }
    // Every day is assigned to exactly one worker
    rows("coverdays", [d:weekdays]) {
        sum(w:workers) { x[w,d] } | 1
    }
    // Every worker shall get at least one and at most two tasks
    rows("maxtasks", [w:workers]) {
        sum(d:weekdays) { x[w,d] } + y[w] | 2
    }

    solve()

    check("solved", solutionStatus())
    check(4, solutionValue())

    def assignment = [:]
    x().each{ key, val ->
        if (val > 0.5) { assignment[key[1]] = key[0] }
}
    check("bob", assignment["fri"])
    check("carol", assignment["wed"])
}
```