Rima: Building Math Models for Reuse

Geoff Leyland
geoff.leyland@incremental.co.nz

Incremental Limited

ORSNZ 2010

Composing Models from *Reusable* Parts

Reuse: Someone has written a model or model part, and someone would like to use the model or part in a different context

In this talk: You have a model for a *single* knapsack and you would like to extend it to a *multiple* knapsack model (generalised assignment)

One to Many Knapsacks

We would like to fill a single sack with items of the highest value:

```
maximize(sum(i in ITEMS) take(i) * value(i))
sum(i in ITEMS) take(i) * size(i) <= CAPACITY
forall(i in ITEMS) take(i) is_binary
```

One to Many Knapsacks

We would like to fill a single sack with items of the highest value:

```
maximize(sum(i in ITEMS) take(i) * value(i))
sum(i in ITEMS) take(i) * size(i) <= CAPACITY
forall(i in ITEMS) take(i) is_binary
```

One to Many Knapsacks

We would like to fill a single sack with items of the highest value:

```
maximize(sum(i in ITEMS) take(i) * value(i))
sum(i in ITEMS) take(i) * size(i) <= CAPACITY
forall(i in ITEMS) take(i) is_binary
```

One to Many Knapsacks

We would like to fill a single sack with items of the highest value:

```
maximize(sum(i in ITEMS) take(i) * value(i))
sum(i in ITEMS) take(i) * size(i) <= CAPACITY
forall(i in ITEMS) take(i) is_binary
```

Now we'd like to extend the model to cover several sacks:

```
maximize(
  sum(s in SACKS, i in ITEMS) take(s, i) * value(i))
forall(s in SACKS)
  sum(i in ITEMS) take(s, i) * size(i) <= CAPACITY(s)
forall(s in SACKS, i in ITEMS) take(s, i) is_binary
forall(i in ITEMS) sum(s in SACKS) take(s, i) <= 1
```

Why do we care?

It's only a few extra characters, right?

Why do we care?

It's only a few extra characters, right?

- We might have a more complex model than a knapsack
- Altering the text of the model makes it hard to re-use
  - to re-use it, we have to understand it enough to modify it
  - it is hard to share improvements between the two models

Why program by hand in five days what you can spend five years of your life automating?

- Terence Parr

What is Rima?

Rima:

- is *Yet-Another* Math Programming Modelling Language
- focuses on making it easy to construct and re-use models

- is MIT licensed and available at `http://rima.googlecode.com/`
- is implemented in Lua: `http://www.lua.org/`
  - a small, fast "scripting" language
- currently binds to CLP, CBC and lpsolve
- has been submitted to COIN for review

Contents

Algorithms + Data Structures = Programs

- Niklaus Wirth

Symbolic Expressions + Structured Data = Reusable Model Components

Constructing Expressions

Rima expressions involve *references* constructed with `rima.R`:

```
e = rima.R("a") * rima.R("x") + rima.R("b") * rima.R("y")
```

Constructing Expressions

Rima expressions involve *references* constructed with `rima.R`:

```
e = rima.R("a") * rima.R("x") + rima.R("b") * rima.R("y")
```

Expressions are stored symbolically and print nicely:

```
print(e)                                  --> a*x + b*y
```

Constructing Expressions

Rima expressions involve *references* constructed with `rima.R`:

```
e = rima.R("a") * rima.R("x") + rima.R("b") * rima.R("y")
```

Expressions are stored symbolically and print nicely:

```
print(e)                              --> a*x + b*y
```

All the `rima.R`'s are cumbersome, so there is a shortcut:

```
rima.define("a, x, b, y")
e = a * x + b * y
print(e)                              --> a*x + b*y
```

Constructing Expressions

Rima expressions involve *references* constructed with `rima.R`:

```
e = rima.R("a") * rima.R("x") + rima.R("b") * rima.R("y")
```

Expressions are stored symbolically and print nicely:

```
print(e)                                   --> a*x + b*y
```

All the `rima.R`'s are cumbersome, so there is a shortcut:

```
rima.define("a, x, b, y")
e = a * x + b * y
print(e)                                   --> a*x + b*y
```

You can manipulate expressions like references:

```
print(3 * e)                               --> 3*(a*x + b*y)
print(e^2)                                 --> (a*x + b*y)^2
```

Evaluating Expressions

`rima.E` evaluates expressions by matching references to a *table* of values:

```
rima.define("a, x, b, y")
e = a * x + b * y
print(rima.E(e, {a=2,x=3,b=4,y=5}))--> 26
```

Evaluating Expressions

`rima.E` evaluates expressions by matching references to a *table* of values:

```
rima.define("a, x, b, y")
e = a * x + b * y
print(rima.E(e, {a=2,x=3,b=4,y=5}))--> 26
```

If some references are undefined, `rima.E` returns a new expression:

```
print(rima.E(e, {a=2,b=4}))         --> 2*x + 4*y
```

Evaluating Expressions

rima.E evaluates expressions by matching references to a *table* of values:

```
rima.define("a, x, b, y")
e = a * x + b * y
print(rima.E(e, {a=2,x=3,b=4,y=5}))--> 26
```

If some references are undefined, rima.E returns a new expression:

```
print(rima.E(e, {a=2,b=4}))         --> 2*x + 4*y
```

The values you provide as data to rima.E can be other expressions:

```
rima.define("xpos, xneg")
print(rima.E(e, {x=xpos - xneg})    --> a*(xpos - xneg) + b*y
```

A Simple LP (1)

rima.mp.new creates a model, and rima.mp.C builds a constraint:

```
rima.define("a, b, x, y")

M = rima.mp.new({
  sense = "maximise",
  objective = a*x + b*y,
```

A Simple LP (1)

rima.mp.new creates a model, and rima.mp.C builds a constraint:

```
rima.define("a, b, x, y")

M = rima.mp.new({
  sense = "maximise",
  objective = a*x + b*y,

  C1 = rima.mp.C(x + 2*y, "<=", 3),
  C2 = rima.mp.C(2*x + y, "<=", 3),
```

A Simple LP (1)

`rima.mp.new` creates a model, and `rima.mp.C` builds a constraint:

```
rima.define("a, b, x, y")

M = rima.mp.new({
  sense = "maximise",
  objective = a*x + b*y,

  C1 = rima.mp.C(x + 2*y, "<=", 3),
  C2 = rima.mp.C(2*x + y, "<=", 3),

  x = rima.positive(),
  y = rima.positive()
})
```

A Simple LP (2)

As with expressions, you can print the model:

```
print(M)
--> Maximise:
-->   a*x + b*y
--> Subject to:
-->   C1: x + 2*y <= 3
-->   C2: 2*x + y <= 3
-->   0 <= x <= inf, x real
-->   0 <= y <= inf, x real
```

A Simple LP (3)

`rima.mp.solve` takes the model and a table of data and solves:

```
primal, dual = rima.mp.solve("clp", M, {a=2, b=2})
print(primal.objective)                 --> 4
print(primal.x)                         --> 1
print(primal.y)                         --> 1
print(primal.C1)                        --> 3

print(dual.x)                           --> 0
print(dual.C1)                          --> 0.333
```

`M` **encapsulates a complete, symbolic representation of the model**

Arrays, Sums and Array Assignment

You can index references as if they were arrays:

```
rima.define("X")
e = X[1] + X[2] + X[3]
print(e)                                   --> X[1] + X[2] + X[3]
```

Arrays, Sums and Array Assignment

You can index references as if they were arrays:

```
rima.define("X")
e = X[1] + X[2] + X[3]
print(e)                           --> X[1] + X[2] + X[3]
print(rima.E(e, {X={1,2,3}}))      --> 6
```

Arrays, Sums and Array Assignment

You can index references as if they were arrays:

```
rima.define("X")
e = X[1] + X[2] + X[3]
print(e)                              --> X[1] + X[2] + X[3]
print(rima.E(e, {X={1,2,3}}))         --> 6
```

`rima.sum` sums an expression over a set:

```
rima.define("x, X")
e = rima.sum{x=X}(x^2)
print(rima.E(e, {X={1,2,3}}))         --> 14
```

Arrays, Sums and Array Assignment

You can index references as if they were arrays:

```
rima.define("X")
e = X[1] + X[2] + X[3]
print(e)                              --> X[1] + X[2] + X[3]
print(rima.E(e, {X={1,2,3}}))         --> 6
```

`rima.sum` sums an expression over a set:

```
rima.define("x, X")
e = rima.sum{x=X}(x^2)
print(rima.E(e, {X={1,2,3}}))         --> 14
```

You can assign to a whole array at once:

```
rima.define("i, X")
t = { [X[i]] = 2^i }
print(rima.E(X[5], t))                --> 32
```

Structures

As well as using arrays, you can also index references as if they were structures:

```
rima.define("item")
mass = item.volume * item.density
print(mass)
  --> item.volume * item.density
print(rima.E(mass, {item={volume=10, density=1.032}}))
  --> 10.32
```

A Structured Knapsack (1)

```
rima.define("i, items")        -- items in knapsack
rima.define("capacity")

knapsack = rima.mp.new()

knapsack.sense = "maximise"
knapsack.objective = rima.sum{i=items}(i.take * i.value)

knapsack.capacity_limit = rima.mp.C(
  rima.sum{i=items}(i.take * i.size), "<=", capacity)

knapsack.items[{i=items}].take = rima.binary()
```

Remember this model, because we won't change it from here on

A Structured Knapsack (1)

```
rima.define("i, items")         -- items in knapsack
rima.define("capacity")

knapsack = rima.mp.new()

knapsack.sense = "maximise"
knapsack.objective = rima.sum{i=items}(i.take * i.value)!

knapsack.capacity_limit = rima.mp.C(
  rima.sum{i=items}(i.take * i.size), "<=", capacity)

knapsack.items[{i=items}].take = rima.binary()
```

Remember this model, because we won't change it from here on

A Structured Knapsack (1)

```
rima.define("i, items")       -- items in knapsack
rima.define("capacity")

knapsack = rima.mp.new()

knapsack.sense = "maximise"
knapsack.objective = rima.sum{i=items}(i.take * i.value)

knapsack.capacity_limit = rima.mp.C(
  rima.sum{i=items}(i.take * i.size), "<=", capacity)

knapsack.items[{i=items}].take = rima.binary()
```

Remember this model, because we won't change it from here on

A Structured Knapsack (1)

```
rima.define("i, items")        -- items in knapsack
rima.define("capacity")

knapsack = rima.mp.new()

knapsack.sense = "maximise"
knapsack.objective = rima.sum{i=items}(i.take * i.value)

knapsack.capacity_limit = rima.mp.C(
  rima.sum{i=items}(i.take * i.size), "<=", capacity)

knapsack.items[{i=items}].take = rima.binary()
```

Remember this model, because we won't change it from here on

A Structured Knapsack (2)

As usual, rima can write the model out for us:

```
print(rima.repr(knapsack, {format="latex"}))
```

In LaTeX:

$$\begin{aligned}
\textbf{maximise} \quad & \sum_{i \in \text{items}} i_{\text{take}} i_{\text{value}} \\
\textbf{subject to} \quad & \\
\text{capacity_limit} : \quad & \sum_{i \in \text{items}} i_{\text{size}} i_{\text{take}} \leq \text{capacity} \\
& \text{items}_{i,\text{take}} \in \{0, 1\} \forall i \in \text{items}
\end{aligned}$$

A Structured Knapsack (3)

```
ITEMS = {
  camera   = { value =  15, size =  2 },
  necklace = { value = 100, size = 20 },
  vase     = { value =  15, size = 20 },
  picture  = { value =  15, size = 30 },
  tv       = { value =  15, size = 40 },
  video    = { value =  15, size = 30 },
  chest    = { value =  15, size = 60 },
  brick    = { value =   1, size = 10 }}

primal = rima.mp.solve("cbc", knapsack,
  {items=ITEMS, capacity=102})

print(primal.objective)           --> 160
print(primal.items.camera.take)   --> 1
print(primal.items.vase.take)     --> 1
print(primal.items.brick.take)    --> 0
```

Constraints are Data Too

Suppose, for example, you can only pick one of the camera or the vase.

Constraints, like expressions, are just data, so modelling this is easy:

```
primal = rima.mp.solve("cbc", knapsack,
  {items=ITEMS, capacity=102,
  camera_xor_vase =
    rima.mp.C(items.camera.take + items.vase.take, "<=", 1)})

print(primal.objective)              --> 146
print(primal.items.camera.take)      --> 1
print(primal.items.vase.take)        --> 0
```

What if we want to reuse this constrained model?

Extensible Models

`rima.mp.new` can take two arguments, the model you want to extend and any extensions to the model:

```
side_constrained_knapsack = rima.mp.new(knapsack, {
  camera_xor_vase =
    rima.mp.C(items.camera.take + items.vase.take, "<=", 1)})

primal = rima.mp.solve("cbc", side_constrained_knapsack,
  {items=ITEMS, capacity=102})

print(primal.objective)                 --> 146
```

Multiple Sacks

Now we are ready to try a multiple sack model:

```
rima.define("s, sacks")

multiple_sack = rima.mp.new({
  sense = "maximise",
  objective = rima.sum{s=sacks}(s.objective),
  only_take_once[{i=items}] =
    rima.mp.C(rima.sum{s=sacks}(s.items[i].take), "<=", 1)
})
```

Note that:

- we are treating `sacks` like a “substructure”
- we have not said anything about what `sacks` is

Multiple Sacks

Now we are ready to try a multiple sack model:

```
rima.define("s, sacks")

multiple_sack = rima.mp.new({
  sense = "maximise",
  objective = rima.sum{s=sacks}(s.objective),
  only_take_once[{i=items}] =
    rima.mp.C(rima.sum{s=sacks}(s.items[i].take), "<=", 1)
})
```

Note that:

- we are treating `sacks` like a "substructure"
- we have not said anything about what `sacks` is

Multiple Sacks

Now we are ready to try a multiple sack model:

```
rima.define("s, sacks")

multiple_sack = rima.mp.new({
  sense = "maximise",
  objective = rima.sum{s=sacks}(s.objective),
  only_take_once[{i=items}] =
    rima.mp.C(rima.sum{s=sacks}(s.items[i].take), "<=", 1)
})
```

Note that:

- we are treating `sacks` like a "substructure"
- we have not said anything about what `sacks` is

Multiple Knapsacks

We can specify what the knapsack submodel is when we solve:

```
primal = rima.mp.solve("cbc", multiple_sack, {
  items = ITEMS,
  [sacks[s].items] = items,
  sacks = {{capacity=51}, {capacity=51}},
  [sacks[s]] = knapsack})

print(primal.objective)                  --> 146
```

Sack 1: `camera, vase, brick`
Sack 2: `necklace, video`

Multiple *Constrained* Knapsacks

What if we can't carry the camera and vase in the same sack?

```
primal = rima.mp.solve("cbc", multiple_sack, {
  items = ITEMS,
  [sacks[s].items] = items,
  sacks = {{capacity=51}, {capacity=51}},
  [sacks[s]] = side_constrained_knapsack})

print(primal.objective)                    --> 146
```

Sack 1: `camera, picture, brick`
Sack 2: `vase, necklace`

Conclusion

We wrote a knapsack model and reused it without any modification:

- in a side-constrained knapsack
- as a part of a multiple knapsack problem
- as a part of a multiple side-constrained knapsack problem

Structured symbolic models enable reuse *without* alteration:

- we only need to understand the model interface to reuse it
- it is easy to share improvements

Can we "compose" complex models, and is it worthwhile?