Rima: Building Math Models for Reuse

Geoff Leyland
Incremental Limited
geoff.leyland@incremental.co.nz

Abstract

Rima is a yet-another tool for formulating mathematical models. Rima's goal is to make it easy to write reusable models. To this end, it supports composing a model from parts, and makes it easy create generic parts. Models are defined symbolically, problem data is late bound, and models can be highly structured. Rima is implemented in Lua, binds to CLP, CBC and lpsolve, and is awaiting review to become part of COIN.

Keywords: model reuse, submodels, modelling languages, model abstraction

1 Introduction

1.1 Model Reuse

In this paper *reuse* means reusing a model means taking an existing model and reusing it with minimal modifications in a context for which it was not originally intended. Most math modelling languages offer some separation of model equations and data, meaning the same model can be reused with different data, but this is a weak case of reuse, and, frankly, even facilities for separation of equations and data in many languages are weak[1].

In this paper we will focus on a single case of reuse: a model for a single knapsack model is available, and we wish to reuse it in a model that consists of multiple knapsacks. We've chosen this example because a knapsack model is simple enough that the model should not complicate the message of this paper, and because is not possible to reuse a knapsack in this manner in any other modelling language without modifying the original model.

1.2 Reusing a Knapsack

The "traditional" view of a knapsack is that a burglar is running around a house with a bag to carry his or her loot, and wishes to steal items of as much total value as possible, without overfilling the sack.

The knapsack model is presented in pseudocode below:

[1]Most languages' example code puts the model *after* the data

```
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 extend our "burglar" model so that there are several burglars acting together, each with a sack. They'd like to maximise the value of all the sacks, but they can't overfill any sacks. In pseudocode this model is:

```
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
```

The first three lines of the multiple-knapsack model have the same purpose as the three lines of the knapsack model, only we've had to modify every line to add a sacks index (highlighted in italics). The fourth line is new information - an extra constraint that says that just because you have several sacks now, you can't steal the TV twice - making this "multiple knapsack" a generalised assignment problem.

We can't avoid adding the fourth line to the model, as it adds extra information. This paper concerns avoiding the changes to the first three lines.

1.3 What's the big deal?

The changes to the knapsack model only amount to a few characters, so what's the big deal? It's not that hard to edit the model quickly, right?

Firstly, we might be working on a more complex model than a knapsack, and the changes might amount to a few more characters. But the main problem is that editing the model violates two fundamental tools we have to build an understanding of the world.

1.3.1 Abstraction

Abstraction can be viewed as the separation of *implementation* and *interface*, and is a fundamental tool for understanding our world. In order to build a car, it is necessary to have detailed understanding of areas such as internal combustion engines, steering systems, suspension, and brakes[2]. In order to drive a car, though, you only need to be able to turn a wheel, push two or three pedals, and have a rudimentary understanding of the road rules[3]. This separation of implementation and interface is what allows us to drive cars without a Ph.D. in thermodynamics[4].

In order to reuse the knapsack model in a multiple knapsack, we had to edit it, which required us to understand its implementation, violating abstraction, and making reuse unnecessarily complicated.

[2]Understanding brakes recently appears to have become optional for some vehicle manufacturers

[3]Very rudimentary if you live in Auckland

[4]I don't deny that it might be a better world if only people who could accurately describe an Otto cycle were allowed to drive

1.3.2 Standing on the Shoulders of Giants

Human knowledge progresses through the sharing of knowledge, and by allowing the whole population to come up with improvements and act as a market for ideas and improvements[5].

If one person writes a knapsack model, and another takes that model and modifies it as shown to work as a multiple knapsack, then sharing improvements becomes much harder. If one or other author finds and fixes a bug, or makes an improvement to the model, that improvement is hard to share, because it's hard to tell which differences are due to the changes that were made because of the differences in the model context, and which are the actual improvement.

Changing the model makes sharing ideas hard, and hinders human progress[6].

In this paper, we will build a single knapsack model, and then reuse it in a multiple knapsack *without making any changes to the original knapsack*. In doing so we will develop a system for building math models by composing reusable parts.

1.4 Introducing Rima

> Why program by hand in five days what you can spend five years of your life automating?
>
> - Terence Parr, author of numerous compiler tools

Faced with the problems described above, I developed Rima, a new modelling tool that focuses on making it easy to construct and re-use models. Rima:

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

1.4.1 Lua

There has been some confusion about the use of Lua in Rima, so it is worthwhile to give the language a brief introduction.

According to lua.org, "Lua is a powerful, fast, lightweight, embeddable scripting language".

For powerful, Lua is a full-featured scripting language with garbage collection, proper closures, proper coroutines, tail-call optimisation and no global interpreter lock.

As far as fast goes, the default implementation of Lua is about 4 times faster than the default implementation of Python on the same problem. The fast, just-in-time compiled version is about 10 times faster than the fastest implementation in Python, and has performance impressively close to C.

[5]Thanks Nicholas Taleb

[6]In the case of the knapsack, we might not mind hindering burglars

For lightweight, the windows version of Lua is about 140kb, and Lua has been ported to systems with as little as 64kb of RAM.

Lua is very easy to bind to, and is embedded a large number games, such as World of Warcraft and Civilisation 5, is in commercial software such as Adobe Lightroom and in software tools like Nmap, and Apache.

However, this paper is *not* about Lua, it is about methods building reusable math models that could work in any language, and for these purposes, Lua is no more than an implementation detail[7].

2 Symbolic Expressions

2.1 Expressions

A mathematical model is built from a set of mathematical expressions - the expression for the objective, and the expressions making up the constraints. In Rima, the objective and constraints are stored as symbolic expressions. This means the expressions are independent of any data, and particularly the dimensions of any sets, and provides strong separation of the model and the model data.

Dedicated modelling languages such as AMPL and GAMS do the same, so we are not introducing anything new. However, in most "language bindings" to modelling systems, such as PuLP and FlopC++, the equations you build are directly manipulating matrix rows.

2.1.1 Constructing Expressions

Rima expressions involve *references*, which are placeholders for variables whose values we'll look up later. References are constructed with `rima.R`, as illustrated below:

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

Expressions can be printed, and you can see below that the expression has been stored in symbolic form (`--` introduces a comment in lua, and `-->` is a convention we'll use in this paper for showing output):

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

All the `rima.R`'s become cumbersome, so `rima.define` provides a shortcut by letting us define references in advance:

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

Expressions don't just have to be constructed from references, they can be about other expressions too:

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

[7] A very carefully chosen implementation detail!

In all these cases, `e` encapsulates a symbolic representation of the expression, providing one component of a clear separation of expressions and data.

2.1.2 Evaluating Expressions

Combining an expression with data makes it concrete (unless some references are undefined, see below) so it can be evaluated. `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 involving the undefined references:

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

The values you provide as data to `rima.E` are not restricted to being immediate data, they can be other expressions:

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

2.2 A Simple LP

Expression construction is just enough to allow us to build a very simple optimisation model.

First, we declare some references:

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

Then we create a new modelling environment with `rima.mp.new`. In this case, `rima.mp.new` takes a single argument which is a table of key-value pairs defining parts of the model:

```
M = rima.mp.new({
```

First, we define the objective and its sense (Rima also understands "maximize" and "MaXImiSE"):

```
  sense = "maximise",
  objective = a*x + b*y,
```

Then we define a couple of constraints with `rima.mp.C`. Note that the constraints are named. The constraint construction syntax is a little awkward (the constraint comparison operators are strings) because we have to keep the Lua parser happy:

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

Finally, we define some bounds on the references that will become our LP variables.

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

Rima makes no distinction between "parameters" and "variables" in the way other modelling languages do, but when the time comes to solve, the solver needs to know the bounds of the variables it will solve for.

`M`, the value returned from `rima.mp.C` encapsulates a complete symbolic representation of our little model.

As with expressions, `M` can be printed:

```
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
```

The output is very useful for documentation and debugging.

Of course, we'd like to solve `M` even more than print it. `rima.mp.solve` takes the model and a table of data and solves, returning tables of primal and dual variables:

```
primal, dual = rima.mp.solve("clp", M, {a=2, b=2})
```

The `primal` and `dual` tables are structured in the same way as the input data and model. Because the constraints are named it's easy to access their values and duals:

```
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
```

3 Structured Data

Rima data can be richly structured. Like all other languages we're aware of (modelling and general-purpose), Rima supports arrays. Rima also supports structures. Structures are missing from a number of popular modelling languages, and are poorly supported in others. Likewise, though the languages bindings are written in support structures, the modelling systems themselves are not fully integrated with structures.

3.1 Arrays, Sums and Array Assignment

If you define a reference, and then treat it like an array, Rima will hope that the data you match to the reference is, in fact, an array[8]:

[8] If you give Rima the wrong type of data, it'll try to help you work out what's wrong

```
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. Here, Rima runs through each element of `X`, assigning the current element of `X` to `x` in each iteration of the sum:

```
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, much like the `foreach` syntax you might find in other languages. Here, the `i`th element of `X` is set to 2^i. As with constraints, the syntax is a little awkward: we need to keep the Lua parser happy:

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

3.2 Structures

Like arrays, if you treat a reference like a structure by acting as if it has fields to access, Rima will comply:

```
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
```

Although this seems almost trivial, it's a very uncommon feature in math modelling languages, and it's one of the key features that allows us to address submodels.

3.3 A Structured Knapsack

We now have enough tools to revisit our knapsack model, but this time we'll build the model with structured data.

First, we define some references:

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

Then we construct the new model. Note that here we construct the model with no fields, and then add the fields to it, in contrast to the earlier example where the entire model was specified in the argument to `rima.mp.new` :

```
knapsack = rima.mp.new()
```

Next, we define the objective. Note than we refer to the items as if `take` and `value` are fields of an item, rather than as `take[i]` and `values[i]`:

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

Then we declare the capacity constraint, again, using `i` as if it's a structure:

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

Finally, we set all the `take` variables to binaries, using the array assignment syntax:

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

Remember this model, because now we've written our single knapsack, we won't change it at all.

As before, Rima can print the model:

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

In LaTeX, if we ask nicely:

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

To solve the model we first define the items in the sack. Note that each item is a little table[9]:

```
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 }}
```

Then we solve the model with CBC, passing it the set of items and the capacity of the sack as data:

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

We manage to steal 160 worth of stuff, including the camera and the vase, but not the brick:

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

[9] or dictionary, record, struct or object depending on your upbringing

4 Reusable Model Components

Now that we've built our knapsack model, we'll reuse it in a number of contexts without modifying it at all. We will

- Configure the knapsack in a stronger manner than is currently possible
- Extend (or make a subclass of) the knapsack
- Include the knapsack multiple times in another model

4.1 Customising the Knapsack

Suppose, for example, the burglars have trouble taking both the camera and the vase. In Rima, constraints, like expressions, are just data, and so, as we saw earlier, we can just include the extra constraint in the model data we pass to `rima.mp.solve`:

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

Now the burglars don't take the vase (at a cost of 14):

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

4.2 Extending the Knapsack

What if the burglars keep having the same problem with vases and cameras? They don't want to keep specifying the extra constraint on the command-line. Instead, they'd rather have a new model that included the extra constraint.

For this purpose, `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)})
```

And you can solve the new model exactly as we solved the original knapsack:

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

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

`side_constrained_knapsack` is a model object exactly like the original knapsack - it encapsulates a symbolic model of side-constrained knapsack. We've reused a model in a slightly different context without modifying the original model at all, so we're nearly done.

4.3 Multiple Sacks

Finally, we are ready to try a multiple sack model. Initially, we don't consider that they're knapsacks, and just write a model for burglars with more than one sack. The burglars wish to maximise the value of all sacks, but still can't steal the TV more than once just because they have more than one sack:

```
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 haven't said anything about what the sacks actually are - that'll come later. Also note that we're treating `sacks` and `s` as if they're structures - we're referencing `sacks[s].objective` and `sacks[s].items[i].take`. The ability to name and address submodels is derived from Rima's support for structures, and is a key part of Rima's ability to handle submodels so comprehensively.

We don't need to specify what the sack submodel is until we actually solve the problem (though we could specify it earlier). Here, we solve a multiple knapsack model - the highlighted line showing where we specify the submodel:

```
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 contains the `camera`, `vase` and `brick` while sack 2 contains the `necklace` and `video`.

Now we've achieved our goal - we've used our single knapsack model in a multiple sack model without changing the original model at all.

4.4 Multiple Side-Constrained Knapsacks

As a last trick, remember that the burglars can't carry the camera and vase in the same sack. It's easy to model this: we just change the definition of `sacks` when we solve:

```
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 contains the `camera`, `picture` and `brick` while sack 2 contains the `necklace` and `vase`.

Here we've taken a knapsack model, and without modifying it, we've extended it to be a side-constrained knapsack, and then used it in a multiple sack model. We've also taken a multiple sack model that was intended to be used with a knapsack and, without modifying it, used it with a side-constrained knapsack.

So not only have we achieved model reuse, we've gained a very flexible system for "mix and match" modelling, where we can build models from parts - almost like snapping lego blocks together.

5 Conclusion

To summarise, we wrote a single-knapsack model and reused it without any modification in a side-constrained knapsack, in a multiple knapsack, and a multiple side-constrained knapsack. This was possible because Rima models are symbolic and structured.

Models that are reusable without modification don't violate abstraction, and assist the sharing of modelling knowledge.

The techniques shown clearly work well for simple models such as a knapsack, and we hope to show in the future that the same techniques can work for more complex models.

6 Thanks

- Stu Mitchell, maintainer of PuLP, for clearing the way
- Phil Bishop at the New Zealand Electricity Authority for providing motivation
- Everyone who took the time to review this presentation
- The global financial crisis for giving me plenty of free time last year!