Formalizing optimization problems in the semantic matrix

Peter Schodl

This version: March 14, 2012 21:22

Contents

1 Introduction

This paper assumes knowledge what the SM is and how information is represented in the SM, it only discusses issues that are special for the representation of the OR-Library, or at least for higher-level mathematics.

Acknowledgements. Support by the Austrian Science Fund (FWF) under contract number P20631 is gratefully acknowledged.

2 The OR Library

The Operations Research-Library (OR-Lib), maintained by J. E. Beasley and originally described in [?], is an online resource of test data sets for a variety of Operations Research problems.[1] It contains 111 problem classes, 59 problem classes are downloadable directly from the OR-Lib and 52 are links to data sets outside the OR-Lib. For our project, we concentrate on the 59 problem classes actually contained in the OR-Lib.

The OR-Lib contains data for well-known optimization problems like the travelling salesman problem, the binpacking problem, set covering, Hamlitonian cycle etc. For example, many of the (NP-complete) problems in the seminal work [?] are included in the OR-Library.

For one specific problem, the OR-Lib contains:

- the referece to a publication where this data set was originally described and used
- information about number and size of the files
- a description of the structure of the data in the files
- the data files itself.

[1] The OR-Library is available at `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

3 Representation of semantics in the SM

All semantics is stored in terms of **sems**, i.e, a equation of the form `a.b=c`, where `a`, `b` and `c` are **nodes**. For a definition of the basic concepts and types, see [**?**].

Since writing every single sem into the SM is laborious, we specified a number of commands that take the type of a record to be created and already existing records as input, create the sems according to the type of the record to be created, and returns the new record. This makes feeding information into the SM substantially faster.

3.1 Motivation

We chose a generic way of storing and writing semantic information in the SM. The design was motivated by the following requirements:

- The semantics of a text is essentially given by a parsetree.
- Assuming an incremential parser with access to the SM, this parser must be able to alter and add information for grammatical constructions.
- Not only different types of output must be created (Text, AMPL), but also in different styles: some mathematicians use $\subset$ where others use $\subseteq$, etc. Also, text in different languages can be regarded as different styles of output for the same record in the SM.

We give a small example: Consider a record `#rec` of type `PARAGRAPH`, i.e., `#rec.OP=PARAGRAPH`. This record stores the header of the paragraph in `#rec.header` and the sentences in the body of the paragraph in `#rec.1` ...`#rec.`n. [*** TODO ***]

3.2 Make-command for variables

A variable with optionaly nonempty string ⟨name⟩ (the name of the variable) is created by the make-command `#node = mkvar(`⟨name⟩`)`. This command results in creation of the following record:

`#node.OP = VAR`
`#node.FREE.#node = #node`
and if the string ⟨name⟩ is nonempty, then also
`#node.name = #namenode`
with `VALUE(#namenode) =` ⟨name⟩

3.3 Make-command for constants

A constant with value ⟨string⟩ is created by the make-command `#node = mkconst(`⟨string⟩`)`. This command results in creation of the following record:

`#node.OP = CONST`
`#node.entry = #valuenode`
with `VALUE(#valuenode) =` ⟨string⟩

3.4 Make-command for all other types

All other typed records except records of type `VAR` and `CONST` are created by the make-command `#node = mk(`⟨type⟩,⟨input1⟩,...,⟨input$n$⟩`)` where ⟨input$i$⟩ is either an existing records, or, in curly brackets, a list of records. How this command is interpreted depends on information assumed in the SM. In particular we assume this information to be contained in the record ⟨type⟩`.MKINFO`, which we will simply call `#info`.

For a command `#node = mk(`⟨type⟩,⟨input1⟩,...,⟨input$n$⟩`)` we assume:

`#info.OP = MKINFO`
`#info.NARG =` n
`#info.`i is nonempty for $i = 1, \ldots, n$
and if `#info.`i`.ISLIST` is nonempty for some i, then
`#info.`i`.ISLIST = TRUE`

Given these requirements the command `#node = mk(`⟨type⟩,⟨input1⟩,...,⟨input$n$⟩`)` creates the following record:

`#node.OP =` ⟨type⟩	
`#node.(#info.`i`) =` ⟨input$i$⟩	if `#info.`i`.ISLIST` is empty and `#info.`i `= BINDS`
`#node.BINDS.OP = VARLIST` `#node.BINDS.`⟨bound⟩ `=` ⟨bound⟩	if `#info.`i`.ISLIST` is empty and `#info.`$i \neq$ `BINDS` and ⟨input$i$⟩`.`⟨free⟩ `=` ⟨free⟩
`#node.`k `=` ⟨input$i$⟩$\{k\}$ `#node.NARG =` m	if `#info.`i`.ISLIST = TRUE` and ⟨input$i$⟩ is a cell with m entries
`#node.FREE.OP = VARLIST`	
`#node.FREE.`⟨free⟩j `=` ⟨free⟩	if ⟨input$i$⟩`.FREE.`⟨free⟩ `=` ⟨free⟩

In appendix **??** we give a table that comprises the syntax of the make-commands for all other types.

3.5 Writing information for make-commands

In the last subsection we saw that a general make-command assumes a special record to be able to interpret the input as desired. There is a special make-command which creates such an assumed record, called `mkmkinfo` since it is a make-command for the information for the general make-command.

Again, we abbreviate ⟨type⟩`.MKINFO` by `#info`. Then the command `mkmkinfo(`⟨type⟩,⟨input1⟩,...,⟨input$n$⟩`)` creates the following record:

`#info.OP = MKINFO`	
`#info.NARG =` n	
`#info.`i `=` ⟨input$i$⟩	if ⟨input$i$⟩ is a node
`#info.`i`.ISLIST = TRUE`	if ⟨input$i$⟩ is a cell

4 Generating the output

From this information in the SM, output can be created. This is done by a relatively simple algorithm (less than 200 lines of MATLAB-code), with aid of information stored in the SM. [*** include lines of code ***]

The algorithm works recursively, i.e., if it is called to roduce output for some record, it calls the algorithm again for all children, and connects them by text determined from the type of the record.

Example: For example, consider the expression $7 + 5 = 12$, represented as:

```
#h.RHS = 12
#h.LHS = term_lhs
#h.OP = EQUAL
term_lhs.1 = 7
term_lhs.2 = 5
term_lhs.OP = PLUS
```

We abbreviate the application of the output-algorithm to record `#rec` by $\mathcal{A}$(`#rec`). The argorithm recursively calls subexpressions until every application of the algorithm is completely evaluated to text:

$\mathcal{A}$(`#h`)	
`$` $\mathcal{A}$(`LHS`) `=` $\mathcal{A}$(`RHS`) `$`	since `#h.OP=EQUAL`
`$` $\mathcal{A}$(`LHS`) `= 12 $`	since `#h.RHS = 12` and `12` is a count
`$` $\mathcal{A}$(`term_lhs`) `+` $\mathcal{A}$(`term_rhs`) `= 12 $`	since `LHS.OP=PLUS`
`$ 7 + 5 = 12 $`	since `term_lhs.1=7` and `term_lhs.2=5`

This, however, assumes a nested structure of text, which is not always true. For example, the expression $\neg\exists x : x > x$ would translate into `not exists an $x$ ...`, but not into `there exists no $x$ ...` or `no $x$ exists...`.

The output of a record is recursively produced by the output of its children. There essentally six different procedures to translate the child of a record into output:

1. Output is a fixed string for the record. E.g., `ELLIPSIS` in textmode is replaced by `\ldots`.
2. Output the result of the procedure applied to a child.
3. Output the result of the procedure applied to a child, but with a fixed string before and after this result.
4. Output the result of the procedure applied to the childs 1,2,...,n but with fixed strings to be put before the whole result, after the whole result, before each child, after each child, between each child, etc.
5. Output the result of the procedure applied to the childs 1,2,...,n of a child of the record, but again with fixed strings to be put before the whole result, after the whole result, before each child, after each child, between each child, etc.

6. Call a special program that produces the output. E.g., every type where the value of a node is to be output, requires such a routine.

4.1 Information about output in the SM

The output-algorithm assumes for each type and each mode a record containing information about the output. This information is written into the SM via a special command for every mode of the form:

`mkformulainfo(`⟨type⟩,⟨style⟩,⟨input1⟩,...,⟨input$n$⟩`)`	for formula-mode
`mktextinfo(`⟨type⟩,⟨style⟩,⟨input1⟩,...,⟨input$n$⟩`)`	for text-mode
`mkamplinfo(`⟨type⟩,⟨style⟩,⟨input1⟩,...,⟨input$n$⟩`)`	for ampl-mode

The result is the following record:

⟨type⟩.⟨modeinfo⟩`.OP =` ⟨modeinfo⟩	where ⟨modeinfo⟩ is `FORMULAINFO`, `TEXTINFO` or `AMPLINFO`
⟨type⟩.⟨modeinfo⟩.⟨style⟩ `= #infostyle`	to be read as an abbreviation
`#infostyle.NARG =` n	
`#infostyle.OP = EXCEPTION`	if $n = 1$ and ⟨input1⟩ is a cell of length 2
`#infostyle.OP = LIST`	else
`#infostyle.`i`.OP = STRING` `VALUE(#infostyle.`i`) =` ⟨input$i$⟩	if ⟨input$i$⟩ is a string
`#infostyle.`i`.OP = INFO_TOKEN` `#infostyle.`i`.token =` ⟨input$i$⟩{1}	if ⟨input$i$⟩ is a cell of length 1
`#infostyle.PROGRAM =` ⟨input$i$⟩{2}	if ⟨input$i$⟩ is a cell of length 2 and ⟨input$i$⟩{2} is string
`#infostyle.`i`.OP = INFO_TOKEN` `#infostyle.`i`.before =` ⟨input$i$⟩{1} `#infostyle.`i`.token =` ⟨input$i$⟩{2} `#infostyle.`i`.after =` ⟨input$i$⟩{3}	if ⟨input$i$⟩ is a cell of length 3
`#infostyle.`i`.OP = INFO_LIST` `#infostyle.`i`.before =` ⟨input$i$⟩{1} `#infostyle.`i`.after =` ⟨input$i$⟩{2} `#infostyle.`i`.beforeeach =` ⟨input$i$⟩{3} `#infostyle.`i`.aftereach =` ⟨input$i$⟩{4} `#infostyle.`i`.beforelast =` ⟨input$i$⟩{5} `#infostyle.`i`.between =` ⟨input$i$⟩{6} `#infostyle.`i`.afterfirst =` ⟨input$i$⟩{7}	if ⟨input$i$⟩ is a cell of length $\geq$ 6 (`#infostyle.`i`.afterfirst` may be empty)
`#infostyle.`i`.OP = INFO_GODOWN` `#infostyle.`i`.enter =` ⟨input$i$⟩{1} `#infostyle.`i`.before =` ⟨input$i$⟩{2}{1} `#infostyle.`i`.after =` ⟨input$i$⟩{2}{2} `#infostyle.`i`.beforeeach =` ⟨input$i$⟩{2}{3} `#infostyle.`i`.aftereach =` ⟨input$i$⟩{2}{4} `#infostyle.`i`.beforelast =` ⟨input$i$⟩{2}{5} `#infostyle.`i`.between =` ⟨input$i$⟩{2}{6} `#infostyle.`i`.afterfirst =` ⟨input$i$⟩{2}{7}	if ⟨input$i$⟩ is a cell of length 2 and ⟨input$i$⟩{2} is a cell of length 7

4.2 The algorithm

The algorithm is called with a record, a mode and a style as arguments. First, the algorithm searches for the output-information for the given type of the record and the desired the mode and style.

Finding out the mode: If the mode is 1 (=text) and there is no information for textmode, then the argorithm enters mode 2 (=formula) and stays in this mode for all childs of the record. If the mode is 3 (=AMPL) it stays in this mode.

Finding out the style: If the record has a child `STYLE`, then the style is taken from this node. If not, then if the algorithm was called with some argument, then this style is used. If not, then the style `default` is used.

But there might be no output-information specified for this style. Then for the actual node, `default` is used, and the childs are called with the style determined above.

4.3 Mathmode / Textmode

`ALTERNATIVE` , `default` (in math-mode):
⟨1⟩

`AND` , `default`:
⟨1⟩ , ⟨2⟩ , ... , ⟨$n-1$⟩ `and` ⟨n⟩

`AND` , `default` (in math-mode):
⟨1⟩ `\wedge` ⟨2⟩ `\wedge` ... `\wedge` ⟨$n-1$⟩ `\wedge` ⟨n⟩

`ABS` , `default` (in math-mode):
`\left|` ⟨entry⟩ `\right|`

`ALTERNATIVEWORD` , `default`:
`(alternative notation:` ⟨entry⟩ `)`

`ASSUMPTION` , `default`:
`we assume` `that` ⟨statement⟩

`ACTION` , `default`:
⟨subject⟩ ⟨verb⟩ ⟨qualification⟩

`BINRELATION` , `default`:
`between` ⟨first⟩ `and` ⟨second⟩

`BRACKET` , `default` (in math-mode):
`\left(` ⟨entry⟩ `\right)`

`CASES` , `default`:
⟨1⟩ , ⟨2⟩ , ... , ⟨$n-1$⟩ `and` ⟨n⟩ `, and` ⟨otherwise⟩ `otherwise`

`CASES` , `default` (in math-mode):
`\begin{cases}` ⟨1⟩ ⟨2⟩ ... ⟨$n-1$⟩ ⟨n⟩ ⟨otherwise⟩ `&` `\text{otherwise}` `\end{cases}`

`CASE` , `default`:
⟨formula⟩ `if` ⟨condition⟩

`CASE` , `default` (in math-mode):
⟨formula⟩ `&` `\text{if~~}` ⟨condition⟩ `\\`

`CONCEPT` , `default`:
⟨adjective⟩ ⟨specification⟩ ⟨entry⟩ ⟨qualification⟩ ⟨symbol⟩

`CLAUSE` , `default`:
⟨1⟩ , ⟨2⟩ , ... , ⟨$n-1$⟩ `and` ⟨n⟩

`CHAIN` , `default` (in math-mode):
⟨firstrel⟩ ⟨1⟩ ⟨2⟩ ... ⟨$n-1$⟩ ⟨n⟩

`CITE` , `default`:
`(see` `\textsc{` ⟨author⟩ `}` `[` ⟨label⟩ `]` `)`

`CARD` , `default` (in math-mode):
`|` ⟨entry⟩ `|`

4.4 Styles

The user has the possibility to give preferences about output style. However, these are overridden if:

- a style is not available for a record
- the output algorithm is called with a specific style as input

4.5 Assigning a style to a record

There is a special command for assigning a certain style to a record. The syntax is `mkstyle(`⟨record⟩,⟨mode⟩,⟨ch
This command adds th following record to the existing node ⟨record⟩:

⟨record⟩`.STYLE.`⟨mode⟩`.`⟨child⟩ `=` ⟨style⟩.

It is necessary to be able to choose a style for a child since there is also output-information for a child of a record.

5 Representation of an optimization problem in the SM

The goal for each problem class in the OR-Library is to represent the semantics of the problem description and the data sets in the semantic matrix. While representing the data sets is trivial, capturing the semantics of an optimization problem is a challange.

The general structure of semantics in the SM is: The type of a node `#n` is stored in `#n.OP`, the free variables in `#n.FREE`, the variables bound in this node in `#n.BINDS`, and auxiliary information that is not part of the semantics, in `#n.PROFILE, #n.NARG` and `#n.STYLE`.

A typical problem from the OR-Lib is a record `#problem` that looks like every other text on the topmost level:

`#problem.OP = DOCUMENTINCLUDE`
`#problem.header` is a record containing the header of the document, and each sem
`#problem.<i>` contains a paragraph.

A first paragraph usually contains assumptions, aggreements, introduces the variables and constants, and gives an interpretion. This was done with a user of the OR-Lib in mind, who is at least vaguely familiar with the problem class. So before the problem is stated in mathematical terms, the variables are introduced with some interpretation, e.g., as processing times or costs, as cardinality of items, etc.

So usually we have a list of sentences in `#problem.1`, hence
`#problem.1.OP = LIST` and each of the entries of the list is a sentence, hence
`#problem.1.<i>.OP = SENTENCE`.

The second paragraph in `#problem.2` usually contains the formal description of the optimization problems by a record with `#problem.2.OP = PROBLEM`. This record specifies the set of variables which are fixed in `#problem.2.given`, the set of variables which may be chosen in `#problem.2.find`, the objective function in `#problem.2.target` and a list of constraints in `#problem.2.constraint`. The information whether the objective finction is to be maximized or minimized is stored in `#problem.2.property`.

The third paragragraph describes the data set. Here, we use a special type `ORDATA`, hence `#problem.3.OP = ORDATA`. It contains the number and names of data sets for this problem in the OR-Library in `#problem.3.nrfiles` and `#problem.3.filenames`. The record `#problem.3.algorithm` contains an algorithm that reads the data file, written in pseudocode. The grammar and semantics of the pseudocode would have to be specified to make the representation rigurous. If one file contains more than one problem instance, then the number of instances is stored in `#problem.3.nrproblems`. It there are choices made that are not described in the data set, then they are stored in `#problem.3.choice`.

6 Example of make-commands

For illustration, we give the make-commands for the representation of one problem. This example consists of 293 lines of code and produces about one and a half page of text. The 20 examples together consist of 3211 lines.

```
fzerotoT = mk('FORMULA',zerotoT);
teqzerotoT = mk('EQUAL',vart,zerotoT);
teqonetoT = mk('EQUAL',vart,onetoT);
s0 = mk('SENTENCE',{itracking});
itproblem = mk('OBJECT',[],problem,[],[],[],[],tracking);
ofitp = mk('QUALIFICATION',[],itproblem);
instofitp = mk('OBJECT',[],instance,[],[],[],[],[],ofitp);
dinstofitp = mk('DEFINED',instofitp);
set1toNofstock = mk('OBJECT',[],set,[],fonetoN,[],[],[],ofstock);
set0toTofmom = mk('OBJECT',[],set,[],fzerotoT,[],[],[],ofmoment);
```

```
nou = mk('OBJECT',[],number,[],[],[],[],[],ofunits);
nouofi =  mk('OBJECT',[],nou,[],[],[],[],[],ofstocki);
nouofidummy = mk('DUMMY',nouofi);
inti = mk('IN',nouofidummy,trindex);
nouofiinti = mk('OBJECT',[],nouofi,[],inti);
intXi = mk('OBJECT',Xi,integer,[],[],[],[],[],[],nouofiinti);
foralliXi = mk('FORALL',intXi,stocki,vari);
valofstock = mk('OBJECT',[],value,[],[],[],[],[],ofstocki);
valofstockatt =  mk('OBJECT',[],valofstock,[],[],[],[],[],atmomentt);
realVitvof =  mk('OBJECT',Vit,real,[],[],[],[],[],[],valofstockatt);
it = mk('VARLIST',{vari,vart});
stockimomentt = mk('LIST',{stocki,momentt});
forallitVit = mk('FORALL',realVitvof,stockimomentt,it);
valofstockatt =  mk('OBJECT',[],valofstock,[],[],[],[],[],atmomentt);
realVitvof =  mk('OBJECT',Vit,real,[],[],[],[],[],[],valofstockatt);
ofti  = mk('QUALIFICATION',[],trindex);
valofti = mk('OBJECT',[],value,[],[],[],[],[],ofti);
votiatmt = mk('OBJECT',[],valofti,[],[],[],[],[],atmomentt);
nIt = mk('OBJECT',It,number,[],[],[],[],[],[],votiatmt);
forallmtIt = mk('FORALL',nIt,momentt,vart);
def1rhs = mk('SETBUCKET',{set1toNofstock,set0toTofmom,foralliXi,forallitVit,forallmtIt});
def1 = mk('DEFINITION',dinstofitp,def1rhs);
s1 = mk('SENTENCE',{def1});
par1 = mk('PARAGRAPH',[],{s1});
inni = mk('IN',nouofidummy,nindex);
nouofiinni = mk('OBJECT',[],nouofi,[],inni);
xidummy = mk('DUMMY',xi);
xigeq = mk('GEQ',xidummy,const0);
xiwith = mk('OBJECT',xi,[],[],xigeq);
let1 = mk('LET',xiwith,[],[],nouofiinni);
xieq0 = mk('EQUAL',xi,const0);
zieq0 = mk('EQUAL',zi,const0);
zieq1 = mk('EQUAL',zi,const1);
case1 = mk('CASE',zieq0,xieq0);
casedist = mk('CASES',{case1},zieq1);
let2 = mk('LET',[],casedist);
slet = mk('SENTENCE',{let1,let2});
```

7 The output of the examples from the OR-Library

Multi-dimensional knapsack.

Let integer N be number of contract , let integer M be number of budget , let c_j be contract volume of project j , let $A_{i,j}$ be matrix of estimated cost , let B_j be vector of available budget and let x_j=1 if project j is selected , and x_j=0 otherwise.

Problem: Given integer N, integer M, vector c ($j = 1, \ldots, N$) , matrix A ($i = 1, \ldots, M$

and $j = 1, \ldots, N$) and vector B ($j = 1, \ldots, N$) , find *binary* $x \in \{0,1\}$ ($j = 1, \ldots, N$) such that

$$\sum_{j=1}^{N} c_j \, x_j$$

is maximal under the constraint $\sum_{j=1}^{N} A_{i,j} \, x_j \leq B_j$ for $j = 1, \ldots, N$.

The OR-Lib contains the files `mknap1.txt` , `mknapcb1.txt` , `mknapcb2.txt` , `mknapcb3.txt` , `mknapcb4.txt` , `mknapcb5.txt` , `mknapcb6.txt` , `mknapcb7.txt` , `mknapcb8.txt` and `mknapcb9.txt`

each containing K problems
to be read with:

```
read K
for k = 1:K
  read N
  read M
  read solution
  for j = 1:N
    read c_j
    for i = 1:M
      read A_{i,j}
    end
    read B_j
  end
end
```

Multi-dimensional knapsack.

Let integer N be number of contract , let integer M be number of budget , let c_j be contract volume of project j for $j = 1, \ldots, N$, let $A_{i,j}$ be estimated cost of budget i for project j for $i = 1, \ldots, M$ and $j = 1, \ldots, N$, let B_i be available amount of budget i for $i = 1, \ldots, M$ and let $x_j = 1$ if project j is selected , and let $x_j = 0$ otherwise for $j = 1, \ldots, N$.

Problem : Given integer N , integer M , vector c , matrix A and vector B find binary vector x such that

$$\sum_{j=1}^{N} c_j \, x_j$$

is maximal under the constraint $\sum_{j=1}^{N} A_{i,j} \, x_j \leq \quad$ for $\quad i = 1, \ldots, M$.

The OR-Lib contains the files `mknap1.txt` , `mknapcb1.txt` , `mknapcb2.txt` , `mknapcb3.txt` , `mknapcb4.txt` , `mknapcb5.txt` , `mknapcb6.txt` , `mknapcb7.txt` , `mknapcb8.txt` and `mknapcb9.txt`

each containing K problems
to be read with:

```
read K
for k = 1 : K
   read N
   read M
   read solution
   for j = 1 : N
      read c_j
      for i = 1 : M
         read A_{i,j}
      end
      read
   end
end
```

.