ConstraintProgramming Extensions.jl

An MOI/JuMP extension for constraint programming

Thibaut Cuvelier — CentraleSupélec (université Paris-Saclay)

What is constraint programming (CP)?

- Way of formulating combinatorial problems
  - CP doesn't really work for continuous problems (exception: Ibex, e.g.)
  - Initial focus in CP was on feasibility, but optimisation is also possible
- Quite different from mathematical optimisation:
  - No duality, no convexity, no linear/continuous relaxation, no differentiability at all
  - More generic set of constraints, not just equations and inequalities
  - Real focus on discrete aspect of the problem
  - Mathematical optimisation's support for combinatorial problems is more an afterthought
- CP has had many successes in operational research:
  - Scheduling, time tabling, resource allocation

An example of CP model: solving Sudokus

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Variables:
  - d_{ij}: digit for cell (i, j), a number between 1 and 9
- Constraints:
  - Known digits (hints)
  - Each digit appears once in each row:
    $$\text{alldifferent}\left(d_{ij}\ \forall j\right), \qquad \forall i$$
  - Each digit appears once in each column:
    $$\text{alldifferent}\left(d_{ij}\ \forall i\right), \qquad \forall j$$
  - Each digit appears once in each block:
    $$\text{alldifferent}\begin{pmatrix} d_{3s,3t} & d_{3s,3t+1} & d_{3s,3t+2} \\ d_{3s+1,3t} & d_{3s+1,3t+1} & d_{3s+1,3t+2} \\ d_{3s+1,3t} & d_{3s+1,3t+1} & d_{3s+1,3t+2} \end{pmatrix}, s \in \{0,1,2\}, t \in \{0,1,2\}$$
- What about your typical MIP model 😜?

ConstraintProgrammingExtensions.jl

- What is the state of CP in Julia?
- What is the state of the package?
- What comes next?
- What is missing in Julia/MOI/JuMP?
- What is enabled with this package?

What is the state of CP in Julia?

- Quite a few CP solvers purely written in Julia!
  - ConstraintSolver.jl
  - JuliaConstraints organisation and its CBLS.jl
  - SeaPerl.jl

  } Based on MOI

  - JuliaIntervals' IntervalConstraintProgramming.jl

  } Based on ModelingToolkit
- Apart from one solver wrapped with this package, no external CP solver wrapped
  - CPLEX CP Optimizer in CPLEXCP.jl

➢ No easy way to write a model for several solver packages

What is the goal of ConstraintProgrammingExtensions.jl?

- This package sits at the same level as MOI: abstraction of solvers
- Major goal: bring the expressive power of CP to MOI and JuMP
  - Have a system that is on (approximate) parity with MiniZinc
  - Wrap most of the constraints of CPLEX CP Optimizer, Gecode, JaCoP, etc.
- Current non-goals:
  - Give access to the whole set of features of CP solvers: exploration tuning, new constraint propagators, callbacks, etc.
  - Provide preprocessing of the formulation, nonnaïve reformulations

Competitors of ConstraintProgrammingExtensions.jl

- MiniZinc / FlatZinc — actively developed (last release in 2021)
  - Dedicated language to describe CP/MIP/SAT models
    - FlatZinc: the bare minimum number of constraints, used to communicate with solvers
    - For instance, no ≥, only ≤
  - MiniZinc can use MIP solvers for CP models
  - “Bridges” when the solver does not support some constraint
    - But not organised as a graph: MiniZinc provides a default implementation (in the hope that the solver supports the new constraints)
    - Each solver can override the rewriting to stop recursion
  - MiniZinc comes with an IDE, a tree visualiser (CP Profiler), a conflict debugger (FindMUS), etc.

Competitors of ConstraintProgrammingExtensions.jl

- Numberjack — more or less actively developed (latest release in 2021)
  - A Python library to build MIP/CP/SAT models
  - Numberjack can convert MiniZinc models as Python files, import and export XCSP models
  - The constraints can be “decomposed” to ease mapping onto solvers
  - For instance, no flexibility in the way models are transformed into MIP, similar to MiniZinc
- Picat — actively developed (latest release in 2021)
  - Functional/declarative programming language, similar to Prolog
  - Library of functions to create CP models

Competitors of ConstraintProgrammingExtensions.jl

- Savile Row — actively developed (last release in 2020)
  - Dedicated language to describe CP/SAT/SMT models (based on Essence-Prime)
  - Reformulations of CP/SAT models to speed up solving times, including techniques to remove symmetry
  - Savile Row can cast CP/SAT models into SMT models, enlarging the available solvers
- OPL — released as part of CPLEX CP Optimizer, actively developed
  - Dedicated language for describe CP/MIP models (with some programming too)
  - CPLEX comes with an IDE (CPLEX Optimization Studio)
  - No reformulations: a MIP model cannot be solved by CPLEX CP Optimizer, and vice-versa

What is the state of the package?

Many standard CP constraints are already available:

- AbsoluteValue
- AllDifferent
- Maximum/Minimum
- BinPacking
- Count, GCC
- Conjunction
- Disjunction
- Knapsack
- Non-overlapping rectangles
- Reification
- Sorting
- Etc.

```
d11 = MOI.add_constrained_variable(
    model, MOI.Integer())
# …

MOI.add_constraint(model,
    MOI.SingleVariable(d11),
    MOI.Interval(1, 9))
# …

MOI.add_constraint(model,
    MOI.VectorOfVariables([d11, d12…]),
    CP.AllDifferent(9))
# …
```

➢ More constraints: easier to model, easier to solve

What is the state of the package?

- One solver is bound:
  - CPLEX CP Optimizer (through its Java API)
- Solver wrappers are typically harder to write for CP solvers
  - Optimisation solvers usually have a callable low-level C API
  - CP solvers mostly have a high-level modelling API, no low-level API, no C
  - Many solvers are written in Java/Scala
- No generic file format to share models among solvers as ubiquitous as LP or MPS
  - MiniZinc, XCSP, AMPL (to some extent), DIMACS (only for SAT): quite high level
  - FlatZinc: low-level variant of MiniZinc
- FlatZinc import and export modules implemented ☺!

What is the state of the package?

- All solvers do not implement all constraints
  - Same problem as with many MOI solvers
  - Same solution: implement bridges
- Many bridges must then be implemented:
  - Between CP sets (some are variants of others, with more parameters)
  - Between CP sets and MIP models
  - So far (July 4): 50 constraint bridges, 6000 lines of code (excluding tests)
  - MOI only has 23 constraint bridges, 5500 lines of code (including more general infrastructure)
  - Far from done…

What is the state of the package?

- To implement some bridges, more information is required about the functions:
  - Does this function have a lower/upper bound? If so, what is this bound?
  - Is it integer, binary?
- Hence, the notion of “trait”
  - It can also be used for function dispatch
- Currently implemented for variable and affine expressions

What comes next?

- In the short term (v0.3):
  - Many more bridges, of course, like #10
    - MiniZinc provides an interesting list of sets to implement (and sometimes bridges)
  - Flesh out the implementation of NLP functions, with function bridges
- In the medium term:
  - More solver wrappers
  - Use this package for Julia CP solvers: #7
- In the long term (v1.0?):
  - More bridges, especially for MIP formulations, like with SOS1 sets or big-M constraints (depending on what the solver proposes): #11, #13, #14, #15
  - SAT models, Boolean algebra as constraints
- In the very long term:
  - Wrap more features of CP solvers, like guiding the exploration or adding new constraints

What is missing in Julia/MOI/JuMP? (1) Non-linearity

- Let's talk about non-linearity…
- So far, in MOI/JuMP, the NLP support is pre-MOI
  - Doesn't play well with MOI (e.g., MOI#1397)
  - Complete rewrite planned
- CP solvers may have specific machinery for constraints like `count(x .== 4) >= 1`
  - How to represent this within the MOI framework?
  - Nonlinear function: `count(x .== 4)`
  - Standard set: `MOI.GreaterThan(1)`
- No need for automatic differentiation, unlike typical NLP

What is missing in Julia/MOI/JuMP? (1) Non-linearity

- Consider this package as a prototype for next-generation NLP support in MOI
- Have a truckload of new AbstractFunction types:
  - `NonlinearScalarAffineFunction`: generalisation to NL terms
  - `NonlinearScalarProductFunction`: also for posynomials (geometric programming)
  - `ExponentialFunction`, `LogarithmFunction`, `CosineFunction`, etc.
- Then, CP-specific functions:
  - `CountFunction`, `ElementFunction` for array indexing, `MaximumFunction`, etc.
- Hugely similar to the way MathOptFormat represents nonlinear functions!

What is missing in Julia/MOI/JuMP? (1) Non-linearity

On the solver side:

- If the combination F-in-S is natively supported: hooray!
- Otherwise:
  - Use function bridges: decompose F-in-S as several constraints

- For instance, `count(x .== 4) >= 1`
  - `CountFunction(x, MOI.EqualTo(4))` –in– `MOI.GreaterThan(1)`
  - `[t, x]` –in– `Count(MOI.EqualTo(4))` and `t` –in– `MOI.GreaterThan(1)`

What is missing in Julia/MOI/JuMP? (2) Variadic parametric types

- Disjunction: OR between several constraints
  - EITHER x ≥ 0 OR y ≥ 0 OR z ≥ 0
  - `[x, y, z]` –in– `Disjunction((MOI.GreaterThan(0), MOI.GreaterThan(0), MOI.GreaterThan(0))`
- Variable number of arguments for Disjunction
  - But they can have different types
  - Julia types cannot have variadic parametric types, only `Tuple` does
  - Hence: parametrise `Disjunction` with a `Tuple`
  - `Disjunction{NTuple{3, MOI.GreaterThan{Int}}}`

What is missing in Julia/MOI/JuMP? (2) Variadic parametric types

- `Disjunction{NTuple{3, MOI.GreaterThan{Int}}}`
- How to dispatch on this thing?
  - Write one function per number of arguments and per type of arguments
  - A lot of code bloat!
  - Or rely on introspection

What is missing in Julia/MOI/JuMP? (3) Structured variables

- Typically, in optimisation solvers, you deal with integers and floats
- Then, what about…
  - Complex numbers? They resemble a pair of floats
  - Time intervals? Again, a pair of numbers
  - Graphs? A larger number of binary variables
  - ➢ Still one VariableIndex for each new type of variable
  - ➢ Still need access to the “subvariables” in some cases (like beginning of time interval)
- ➢ Not just for modelling ease: CP solvers sometimes have graphs as variables!

What is missing in Julia/MOI/JuMP? (3) Structured variables

- Current solution implemented by ComplexOptInterface:
  - Nothing specific when creating variables
  - You cannot have `SingleVariable(z)` –in– `MOI.EqualTo(1 + 2im)`, the code for real variables in MOI is used
- Another solution (MOI#1253):
  - Have `ComplexVariableIndex`, `IntervalVariableIndex`… be composed of two `VariableIndex` (or more)
  - `MOI.add_variable` would take a type argument: scalar real (default), complex, interval…
  - Expressions could be parametrised by the type of variable index: e.g., `ScalarAffineFunction{ComplexVariableIndex, Complex{T}}`

Where do we go from here?

- Most CP sets have MIP bridges: modelling becomes easier for users!
  - E.g., use a bin-packing, circuit, etc. constraint instead of linear constraints
  - Probably not the best models, though
- The new nonlinear infrastructure can be built upon
  - It is probably amenable to DCP
  - However, I make no claim about performance or compatibility with AD systems

How can you help?

- Spread the word for Julia and CP
- Discuss the implementation
- Write new solvers, new solver wrappers and check if all the required features are there
- Benchmark the performance of this package:
  - Compared to MiniZinc to “lower” models
  - Compared to existing JuMP/MOI NLP code
- Write documentation, examples
  - For now, only reference for existing sets