

Julia for Operations Research

Benoît Legat (UCLouvain)

October 22, 2019

Petit Déj' de l'I&D, N-SIDE

JuMP: optimization without compromise

Basic	Advanced
Easy to install, setup, use	Fast
Simple to do simple things	Consistent interface for advanced features
Solver-independent interface	Give access to solver-specific features
Transparently circumvent solver limitations	Allow direct contact with solver
Reliability & consistency	Distributed ecosystem

Usability & Efficiency

Consistency

Flexibility

Modularity

Distributed ecosystem

Usability & Efficiency

Julia language ambition

Initial blog post from 2012:

We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

Multiple dispatch

```
function power_by_squaring(x, p::Integer)
    ...
end
^(x::Number, p::Integer) = power_by_squaring(x, p)
^(x::AbstractMatrix, p::Integer) = power_by_squaring(x, p)
^(x::BigInt, p::Integer) = MPZ.pow_ui(x, p)
^(x::Float64, p::Float64) = ccall("llvm.pow.f64", ...)
^(x::Float32, p::Float32) = ccall("llvm.pow.f32", ...)
```

Julia compiler compiles each method *Just-In-Time* to LLVM IR. LLVM IR is then compiled to machine code. The LLVM IR to machine code layer is shared with all languages relying on LLVM (e.g. Clang, Scala Native).

Multiple dispatch allows both **abstraction** and **specialization**.

Metaprogramming

```
@constraint(model, [i = 1:3, j = 1:i; isodd(i + j)],  
              i*x + j*y + z*(i+j) == i * j)  
  
container(((i, j)->begin  
    expr0 = Val{false}()  
    expr1 = _destructive_add_with_reorder!(expr0, i, x)  
    expr2 = _destructive_add_with_reorder!(expr1, j, y)  
    expr3 = _destructive_add_with_reorder!(expr2, z, i)  
    expr4 = _destructive_add_with_reorder!(expr3, z, j)  
    expr5 = _destructive_add_with_reorder!(expr4, -1.0,  
                                           i, j)  
    con = build_constraint(expr5, EqualTo(0.0))  
    JuMP.add_constraint(model, con, "")  
end, nested(()->1:3, (i,)->1:i;  
              condition=(i, j)->isodd(i + j)))
```

Consistency

A convenient UI should not hide a hacky low-level interface.

- JuMP macros transformed at parsing time to JuMP function calls.
- JuMP function calls redirect to MOI function calls.
- MOI function calls redirect to solver API calls.

Enforcing consistency of solver MOI interfaces

- Abstract Model and Optimizer types.
- Detailed documentation.
- Extensive test suite run by every solvers and MOI internal models.
- Utilities for sanity checks, duplicate removal in functions, get fallbacks...

Flexibility

MathOptInterface (MOI)

- `add_variable(model),`
`add_constrained_variable(model, set).`
- `add_constraint(model, func, set),` e.g. $2x + 3y = 1 \rightarrow$
`(2*x + 3*y)-in-EqualTo(1.0).`
- `set, get` attributes, e.g., `ObjectiveFunction,`
`HeuristicCallback.`
- `submit` submittables, e.g., `HeuristicSolution.`

Extensible framework:

- **Generic** on attribute, function and set types. New ones can be defined **independently**.
- Solver-**specific** features easily exposed to JuMP/MOI users through **custom** attributes, e.g. solver specific callbacks.
- Expose **specialized** problem structure easily through **custom** functions, sets (e.g. indicator constraints, complementarity

Solver-specific callbacks

```
MOI.set(model, GLPK.CallbackFunction(),
        (cb_data) -> begin
            reason = GLPK.ios_reason(cb_data.tree)
            if reason == GLPK.IHEUR
                ...
                accepted = MOI.submit(
                    model, MOI.HeuristicSolution(cb_data), ...)
            elseif reason == GLPK.ICUTGEN
                ...
                MOI.submit(model, MOI.UserCut(cb_data), ...)
            end
        end)
```

Solver-independent callbacks

```
MOI.set(model, MOI.HeuristicCallback(),
        (cb_data) -> begin
            ...
            accepted = MOI.submit(
                model, MOI.HeuristicSolution(cb_data), ...)
        end)

MOI.set(model, MOI.UserCutCallback(),
        (cb_data) -> begin
            ...
            MOI.submit(model, MOI.UserCut(cb_data), ...)
        end)
```

Modularity

Breaking down the MOI backend

Distinction between:

- Unsupported constraints, attributes: resolved by *bridges*.
- Unsupported modification: resolved by a *caching layer*.

MOI.Utilities.CachingOptimizer has 2 modes:

- Automatic: Automatically empty optimizer when a modification is not supported and re-copy at optimize.
- Manual: Need to manually reset optimizer or an error is thrown.

JuMP has 3 modes:

- Direct: No cache nor bridge (can be added manually though).
- Automatic: Cache in AUTOMATIC mode and bridges.
- Manual: Cache in manual mode and bridges.

Semidefinite programming

$$\underset{Q \in \mathcal{S}^n}{\text{minimize}} \quad \langle C, Q \rangle$$

$$\text{subject to} \quad \langle A_i, Q \rangle = b_i$$

$$Q \succeq 0$$

$$\underset{y \in \mathbb{R}^n}{\text{maximize}} \quad \langle b, y \rangle$$

$$\text{subject to} \quad \sum_i A_i y_i \preceq C$$

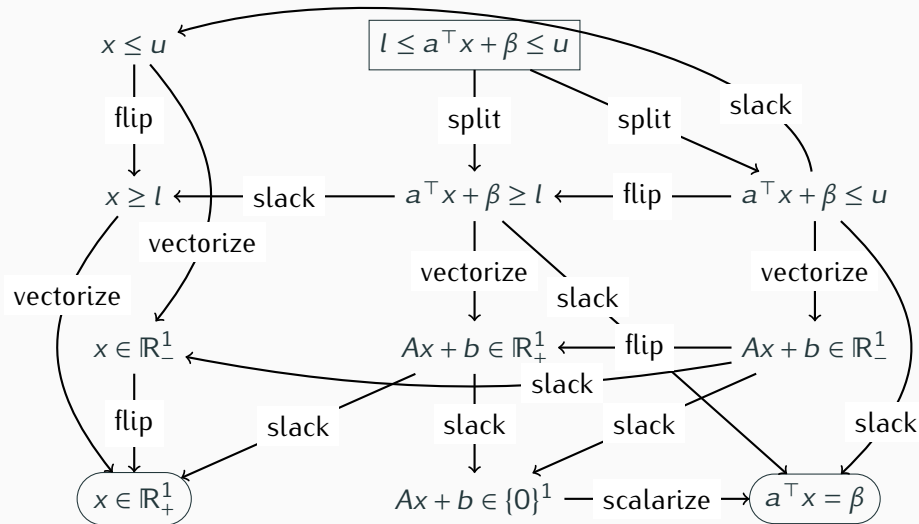
File format: SDPA

Solvers: CSDP, SDPA, DSDP, SDPLR, ...

Variables: Q block diagonal, nonnegative scalar variables (1×1 blocks) or SDP matrices.

Constraints: Affine equations.

Interval constraint for SDPA



Bridges directed hypergraph

Nodes

Node for each set S (variable-in- S).

Node for each constraint F -in- S .

Node for each objective function F .

Edges

For each edge and ingoing node: outgoing nodes are

- variable-in- S created.
- constraints F -in- S created.
- objective F set.

Bridges selection

Multiple dispatch ?

Can we use multiple dispatch where node types are method signatures ?

No. Several bridge can transform same types. Dispatching on more specialized is not accurate.

Need to guide choice to supported types.

Just in Time (JIT) bridge selection

Types F and S are **not limited** to those defined in MOI → **infinitely** many nodes.

Each bridge defined possible **infinitely** many edges.

Need to be **lazy** and compute bridge selection JIT when type first used.

Bridge selection algorithm

Data: Set of bridges B

Invariant: d_V, d_C, d_O are correct when defined.

Given new type: compute types T where d should be defined with Depth First Search (DFS) from the type: $\mathcal{O}(|B|)$.

Compute d at the types using Bellman-Ford algorithm: $\mathcal{O}(|T| \cdot |B|)$.

Distributed ecosystem

MOI wrappers to C/C++ solvers:

- Wrappers developed by JuMP-devs only: BARON, Bonmin, Cbc, CDD, Clp, Couenne, CPLEX, CSDP, ECOS, FICO Xpress, GLPK, Gurobi, Ipopt, SCS, SDPA, SDPT3, SeDuMi, Tulip, PATH
- Co-developed with solver developers: Artelys Knitro, CDCS, MOSEK, NLOpt, OSQP, SCIP

Pure Julia solvers:

- COSMO, ProxSDP, Tulip, Hypatia

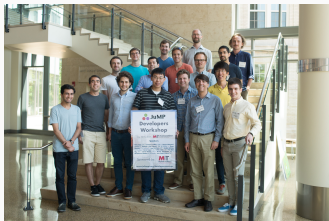
Pure Julia *meta*-solvers:

- Juniper, Alpine, Pavito, Pajarito, Coluna, Dualization

Polynomial optimization:

- With **polynomials**: MomentOpt, PolyJuMP, SumOfSquares
- With **sets**: Polyhedra, SetProg
- **Parametrized**: ParameterJuMP, Parametron
- **Structured**: BlockDecomposition, StructJuMP
- **Stochastic**: SDDP, StochDynamicProgramming, StructDualDynProg
- **Multi-objective**: MultiJuMP, vOptSolver
- PowerModels, BilevelOptimization, JuMPChance, JuMPeR, Convex, NLOptControl

Annual JuMP-dev Workshop



- First at MIT in June 2017.
- Second at Bordeaux before ISMP in June 2018.
- Third in Chile in March 2019.
- Coming at **UCLouvain** between 15th–19th of June 2020.

Videos from past years available on Youtube.