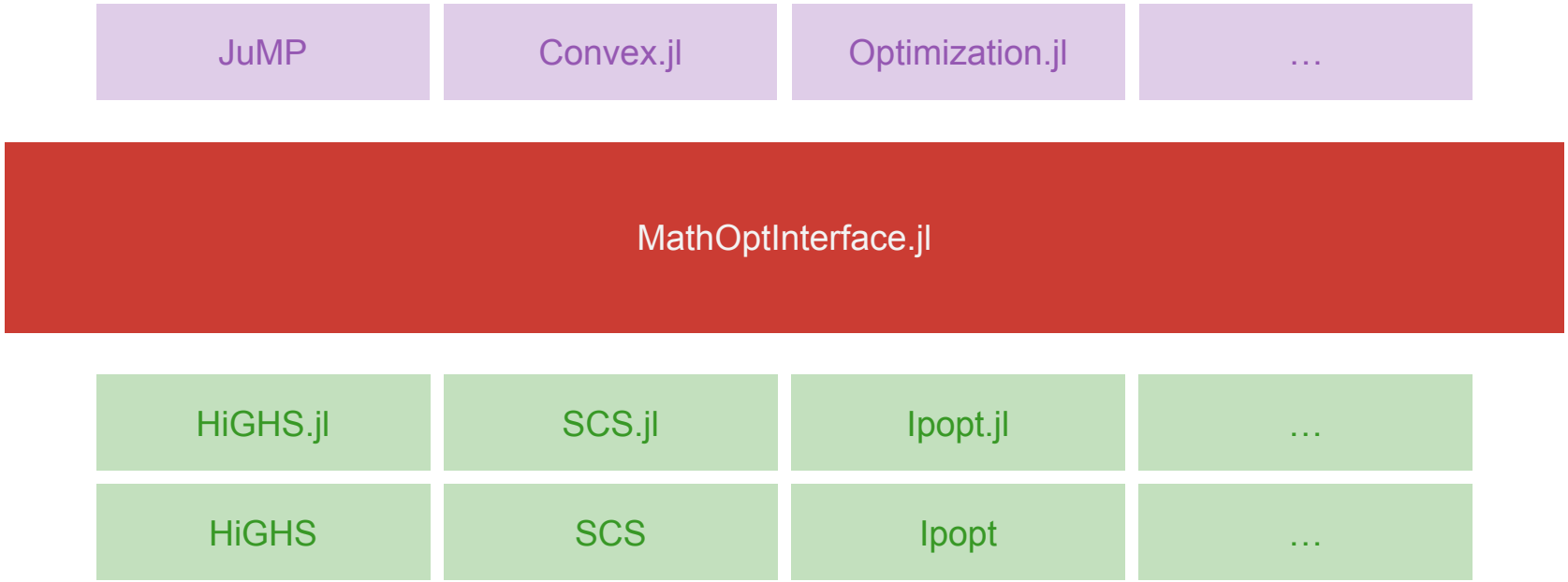
MathOptInterface: a comprehensive overview

Oscar Dowson

JuMP-dev 2025

MathOptInterface.jl

The interface between modeling languages and the solvers



The purpose of this talk

Why is this talk needed:

- MathOptInterface.jl (MOI) is one of the largest packages in all of Julia
- It is the connection between JuMP and solvers
- It uses a novel abstraction
- It has a looooooot of stuff in it

We haven't publicly talked about it much

By the end of this talk you will:

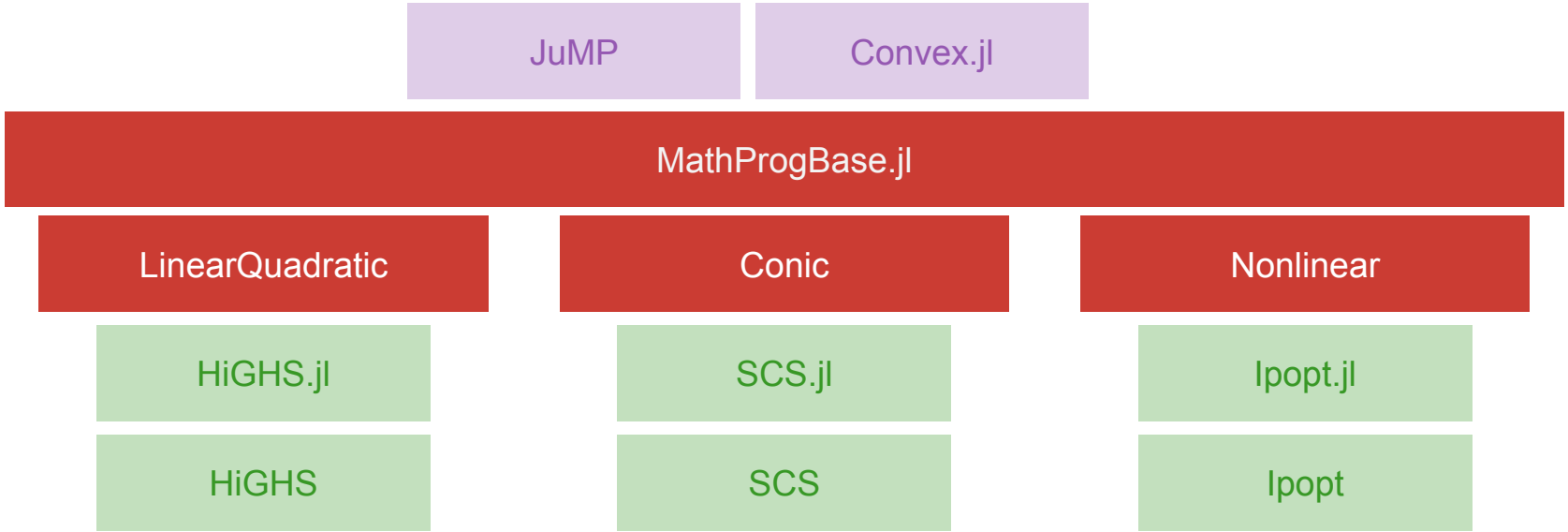
1. Understand the problem we are trying to solve and why we wrote MOI
2. Understand the MOI abstraction
3. Understand what a bridge is and why they are necessary
4. Have an overview of the components in MathOptInterface.jl

By the end of this talk you will not:

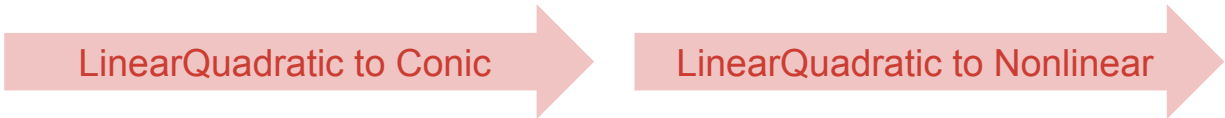
- Be able to write a solver wrapper
- Know how to write code that uses MOI

2013-2019: MathProgBase.jl

MPB divided the world into three problem classes



There were also two “bridges”



2013-2019: Problems with MathProgBase

There were many. Here are four.

Standard forms are not standard

- ECOS and SCS use different orderings for the exponential cone
- Gurobi does not support Ax in Interval
- CSDP does not support free variables

Some solvers mix problem classes

- KNITRO supported nonlinear, but also SecondOrderCone and Complements
- Gurobi was linear quadratic, but now also supports nonlinear

Extending the classes is hard

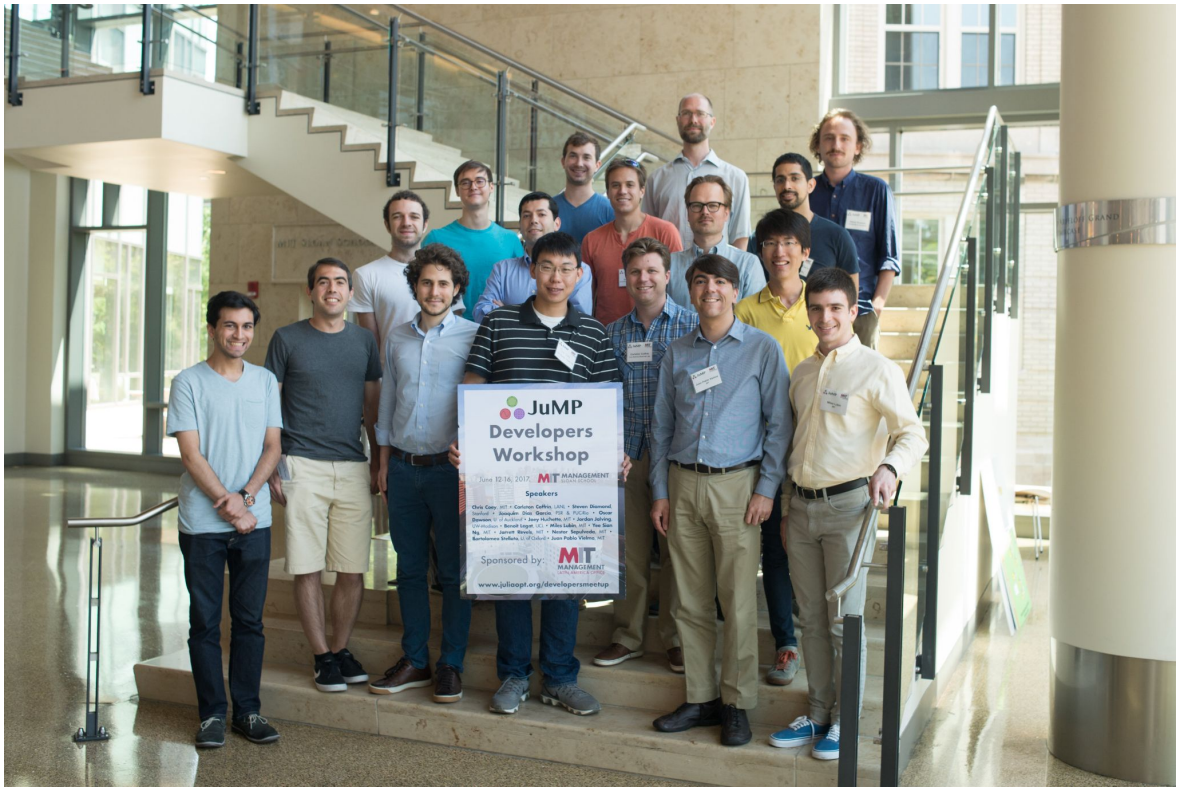
- No indicator constraints
- No complementarity constraints

In-place problem modification was limited

- No support for deleting variables or constraints
- Support for modifying RHS but not LHS

The 2017 JuMP-dev workshop

Release JuMP as 1.0 or rewrite from scratch?



The MathOptInterface standard form

MOI defines a very *regular* standard form

$$\text{minimize: } f_0(x)$$

$$\text{subject to: } f_i(x) \in \mathcal{S}_i \quad \forall\, i \in \{1,\ldots,m\}$$

	Function	Set
`x, Binary`	`VariableIndex(x)`	`ZeroOne()`
`x >= 0`	`VariableIndex(x)`	`GreaterThan(0.0)`
`2x + 1 == y`	`ScalarAffineFunction(2x - y + 0)`	`EqualTo(-1.0)`
`Ax >= b`	`VectorAffineFunction(Ax - b)`	`Nonnegatives()`
`X ≥ 0, PSD`	`VectorOfVariables(vec(X))`	`PositiveSemidefiniteConeSquare()`

Solvers support a subset of functions and sets

HiGHS

Constraints

```
ScalarAffineFunction in {
    EqualTo, GreaterThan, Interval, LessThan
}

VariableIndex in {
    EqualTo, GreaterThan, Interval, LessThan,
    Integer, ZeroOne, Semicontinuous,
    Semiinteger,
}
```

Objective

```
ScalarAffineFunction
ScalarQuadraticFunction
```

Solvers support a subset of functions and sets

SCS

Constraints

VectorAffineFunction in {
    Nonnegatives, Zeros, SecondOrderCone,
    ExponentialCone, DualExponentialCone,
    PowerCone, DualPowerCone, ScaledPSDCone,
    NormNuclearCone, ScaledComplexPSDCone,
    ScaledLogDetConeTriangle,
}

Objective

ScalarAffineFunction
ScalarQuadraticFunction

Solvers support a subset of functions and sets

Ipopt

Constraints

VariableIndex in {
  GreaterThan, LessThan, EqualTo, Interval,
  Parameter,
}

ScalarAffineFunction in {
  GreaterThan, LessThan, EqualTo, Interval,
}

ScalarQuadraticFunction in {
  GreaterThan, LessThan, EqualTo, Interval,
}

Constraints continued…

ScalarNonlinearFunction in {
  GreaterThan, LessThan, EqualTo, Interval,
}

VectorOfVariables in VectorNonlinearOracle

Objective

VariableIndex
ScalarAffineFunction
ScalarQuadraticFunction
ScalarNonlinearFunction

User and solver might speak different formulations

This is a problem. Should users rewrite their model for every solver?

There are multiple ways to write the same constraint

Scalar or vector constraints

- x in GreaterThan(0.0)
- [x] in Nonnegatives(1)
- Ax .== b
- Ax - b in Zeros()

Affine transformations from one set to another

- [t, x, y] in SecondOrderCone(2)
- [t, t / 2, x, y] in RotatedSecondOrderCone(3)

More complicated transforms

- [t, X] in LogDetConeSquare()
- A set of constraints using PSD, ExponentialCone, and LessThan sets

Constraint bridges

Map between equivalent formulations

Each bridge takes as input:

- a function-in-set constraint

and it may:

- add new constraints of a different type
- add new decision variables

Each bridge also implements:

- `MOI.delete(model::MOI.ModelLike, bridge)`
- `MOI.get(model::MOI.ModelLike, ::MOI.ConstraintPrimal, bridge)`
- `MOI.set(model::MOI.ModelLike, ::MOI.ConstraintPrimalStart, bridge, value)`
- `MOI.get(model::MOI.ModelLike, ::MOI.ConstraintDual, bridge)`
- `MOI.set(model::MOI.ModelLike, ::MOI.ConstraintDualStart, bridge, value)`

Bridges let you model unique constraint types

"An affine expression is integer"

```
using JuMP, HiGHS
model = Model(HiGHS.Optimizer)
@variable(model, x)
@constraint(
    model,
    2 * x + 1 in MOI.Integer(),
)
```

```
julia> print_active_bridges(model)
 * Unsupported constraint:
 | MOI.ScalarAffineFunction-in-MOI.Integer
 |  bridged by:
 |   MOIB.Constraint.ScalarSlackBridge
 |  may introduce:
 |   * Supported constraint:
 |     MOI.ScalarAffineFunction-in-MOI.EqualTo
 |   * Supported variable: MOI.Integer

julia> print(unsafe_backend(model))
Feasibility
Subject to:
VariableIndex-in-Integer
 v[2] ∈ ℤ
ScalarAffineFunction-in-EqualTo
 0.0 + 2.0 x - 1.0 v[2] == -1.0
```

Hypergraphs and shortest hyperpaths

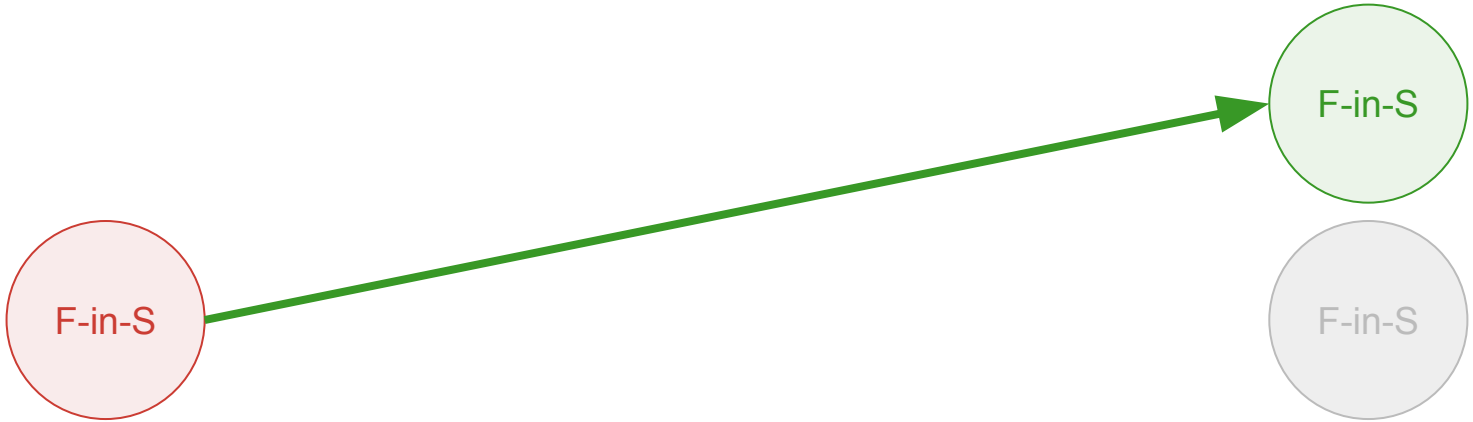
In the ideal case, the solver supports the constraint

User

Solver

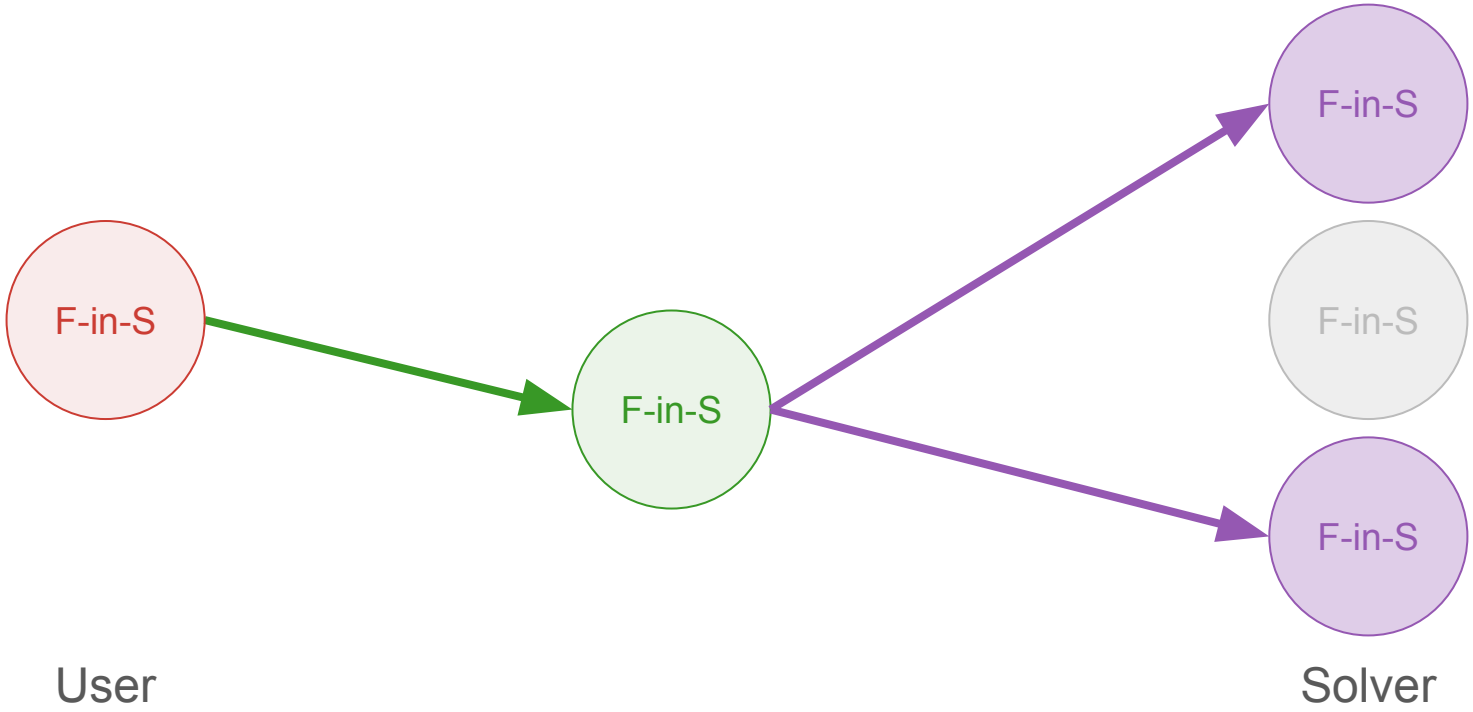
Hypergraphs and shortest hyperpaths

The solver may support the constraint via a single transformation



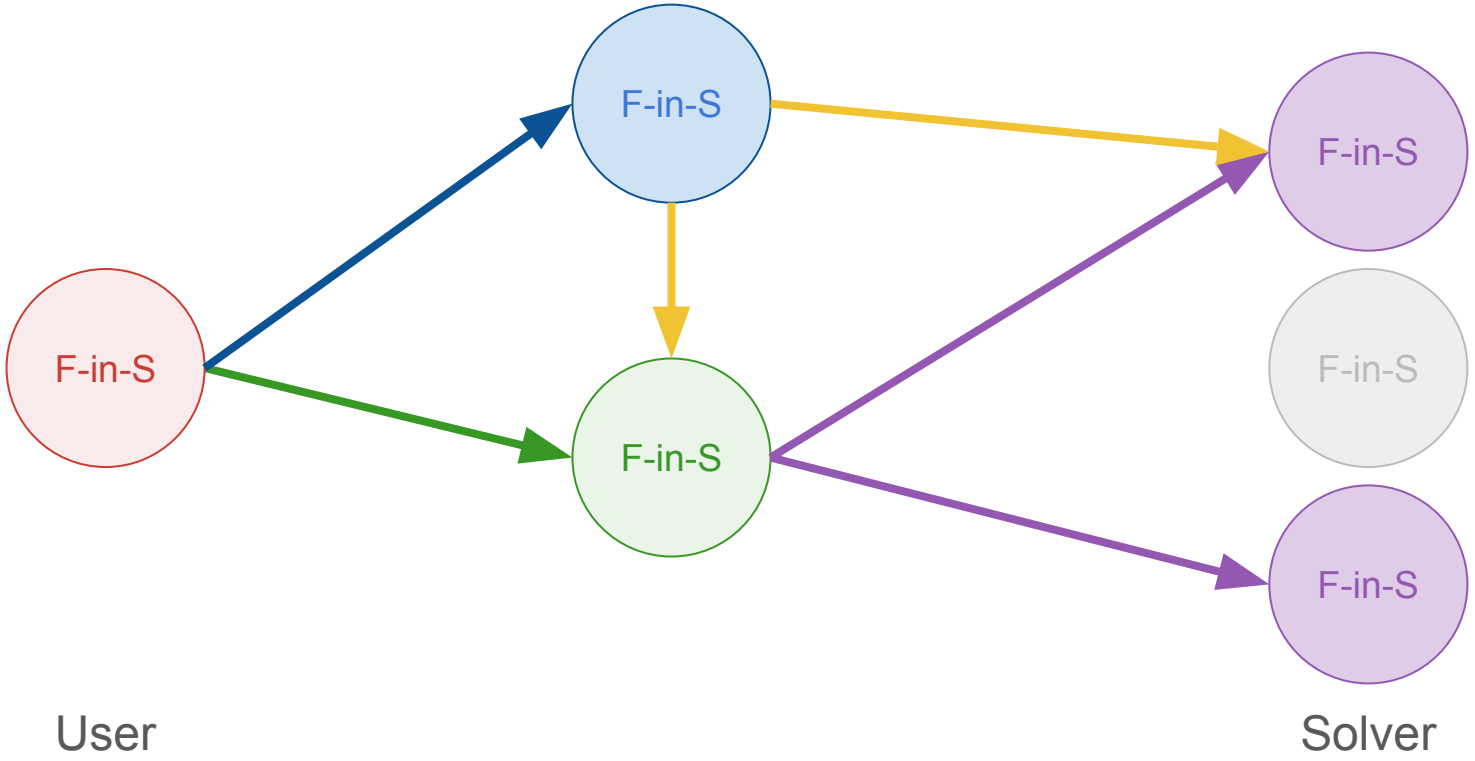
Hypergraphs and shortest hyperpaths

A chain of bridges may be needed, that introduce more than one constraint



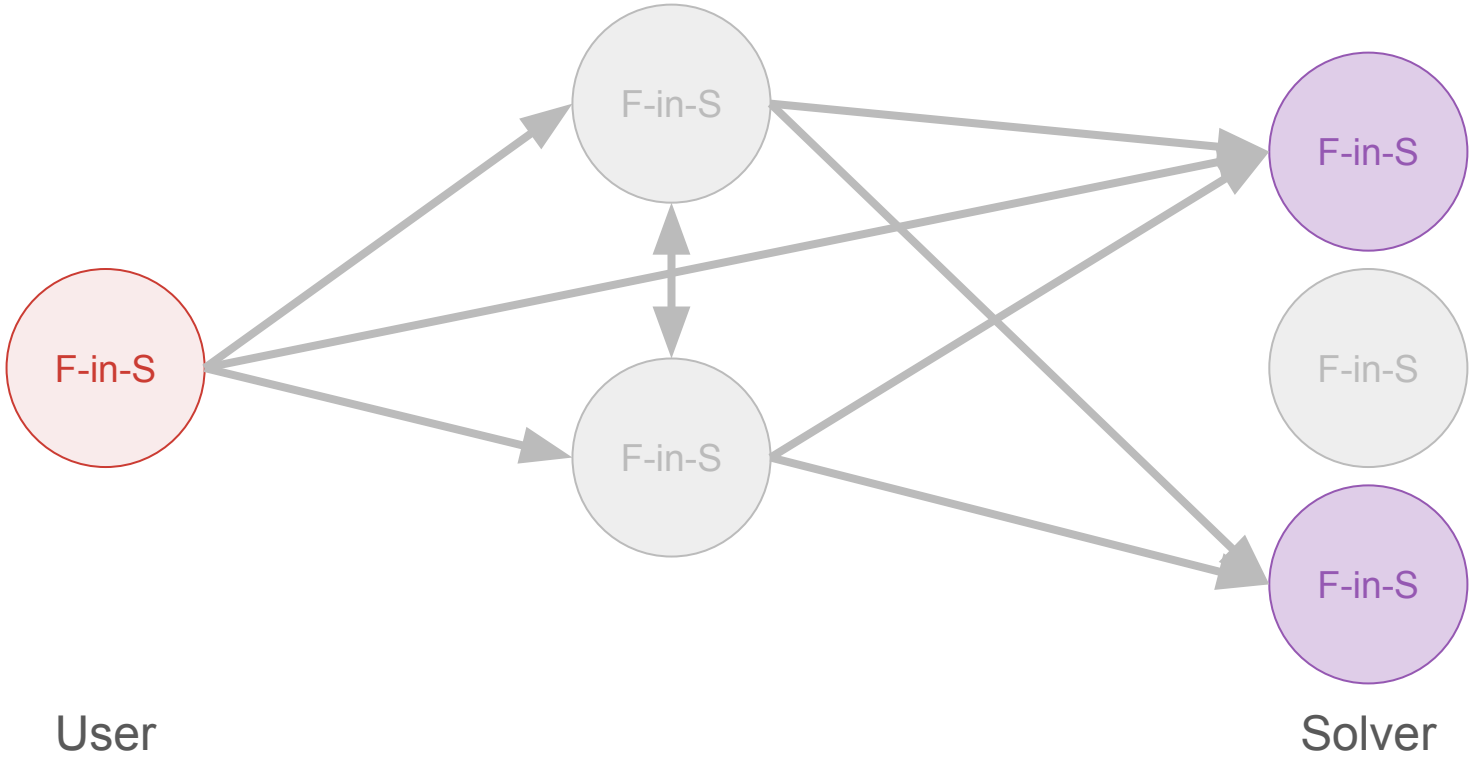
Hypergraphs and shortest hyperpaths

There may be many equivalent paths through the graph



Hypergraphs and shortest hyperpaths

In practice, the graph can be arbitrarily complicated



Hypergraphs and shortest hyperpaths

The optimal bridge is the shortest hyperpath

Let the graph G = (N, E), where:

- N is the set of nodes
- E is the set of bridges

Let S = the set of nodes supported by the solver

Each edge e has:

- A source node s(e) in N
- A set of target nodes T(e) subset N
- A weight w(e)

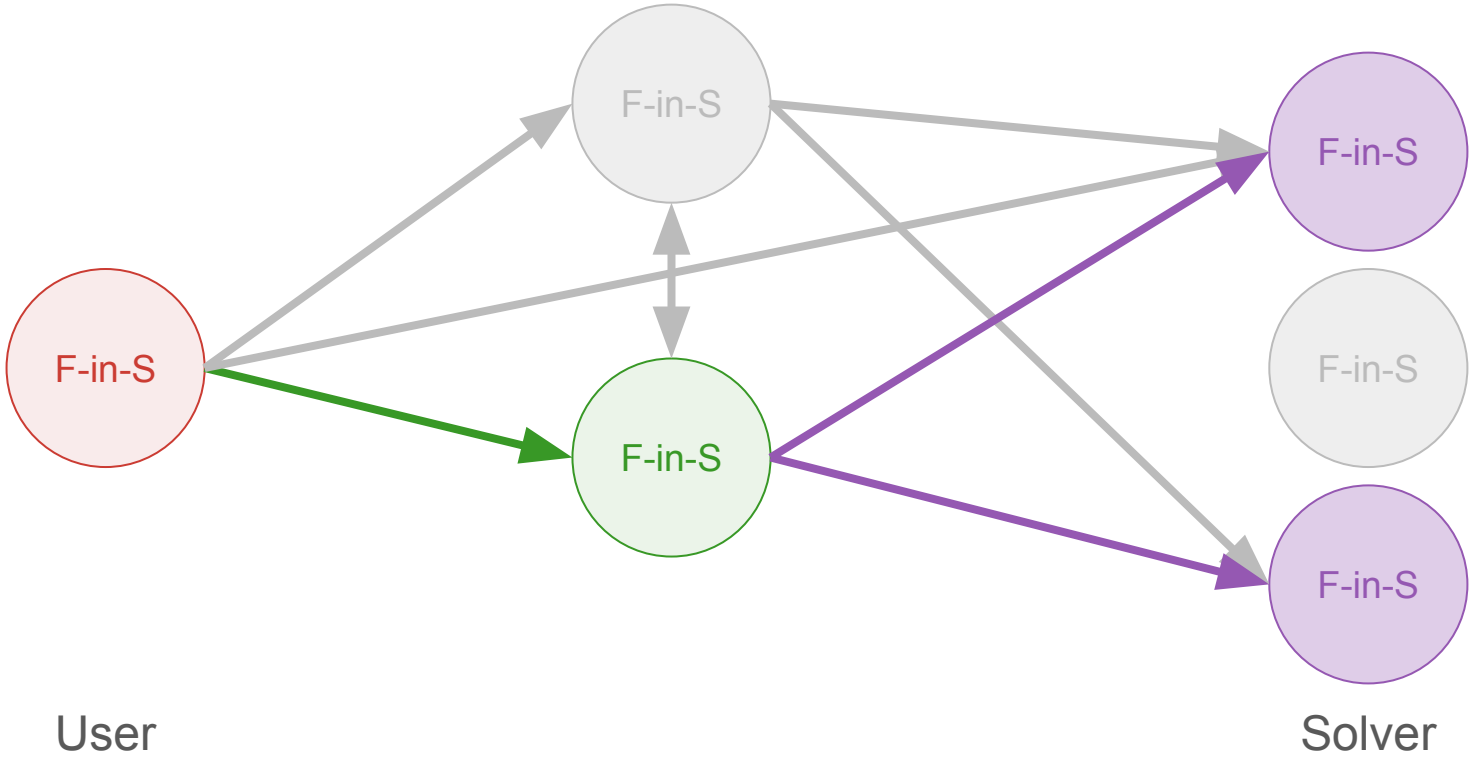
Most bridges choose w(e) = 1. Some bridges use w(e) = 10

Minimum hyperpath is the set of bridges e that minimize:

```
function C(n)
    if n in S
        return 0
    end
    return minimum(
        w(e) + sum(C(m) for m in T(e))
        for e in E if s(e) == n
    )
end
```

Hypergraphs and shortest hyperpaths

The optimal bridge is the shortest hyperpath



There are other types of bridges

a.k.a. it's even more complicated than it seems

Objective bridges reformulate objective functions

User writes

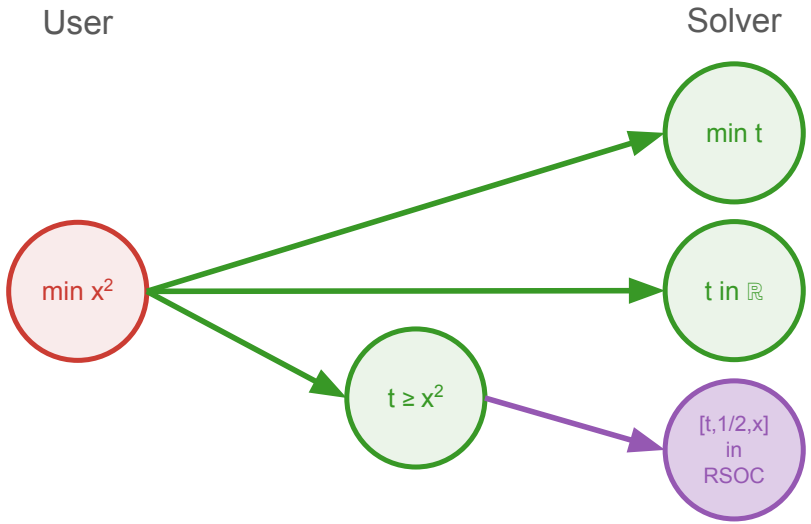
$$\min x^2$$

Bridges.Objective.ScalarSlackBridge

$$\min t: t \geq x^2$$

Bridges.Constraint.QuadToSOC

$$\min t: [t, 1/2, x] \text{ in RotatedSecondOrderCone()}$$



There are other types of bridges

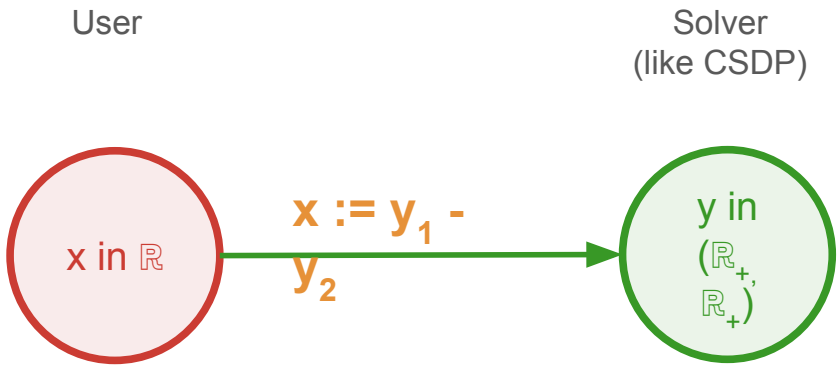
a.k.a. it's even more complicated than it seems

Variable bridges reformulate variable domains

Seems simple. Take a variable -in-set and replace by variable -in- different set with an affine substitution rule.

Every x in the model must be replaced by the substitution rule, and every solution y must be inverted back to the original model.

Keeping track of substitutions is THE most complicated code in all of MOI.



There a

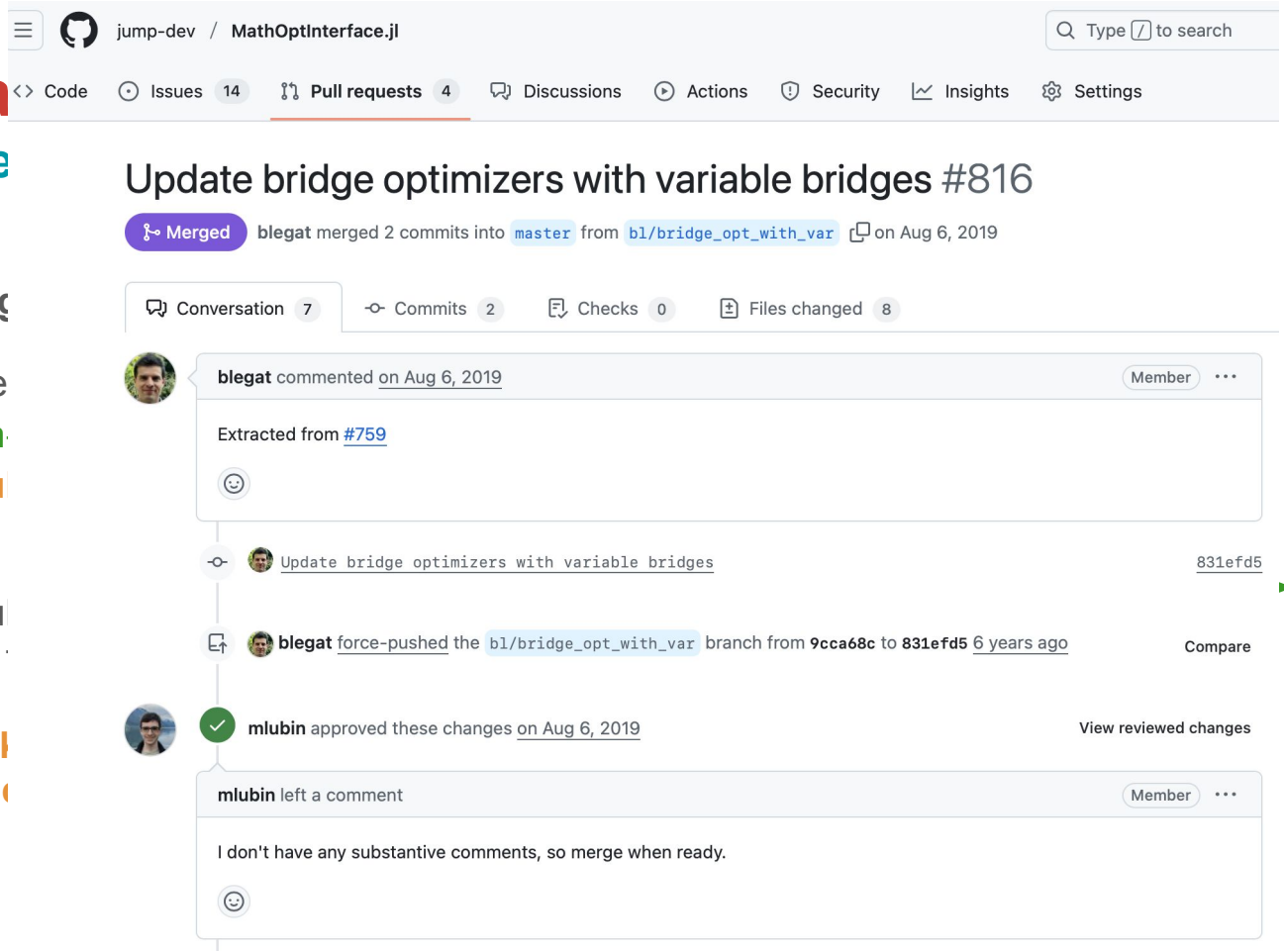
a.k.a. It's e

Variable bridg

Seems simple
by variable -in
substitution ru

Every x in the
substitution ru
inverted back

**Keeping track
complicated**

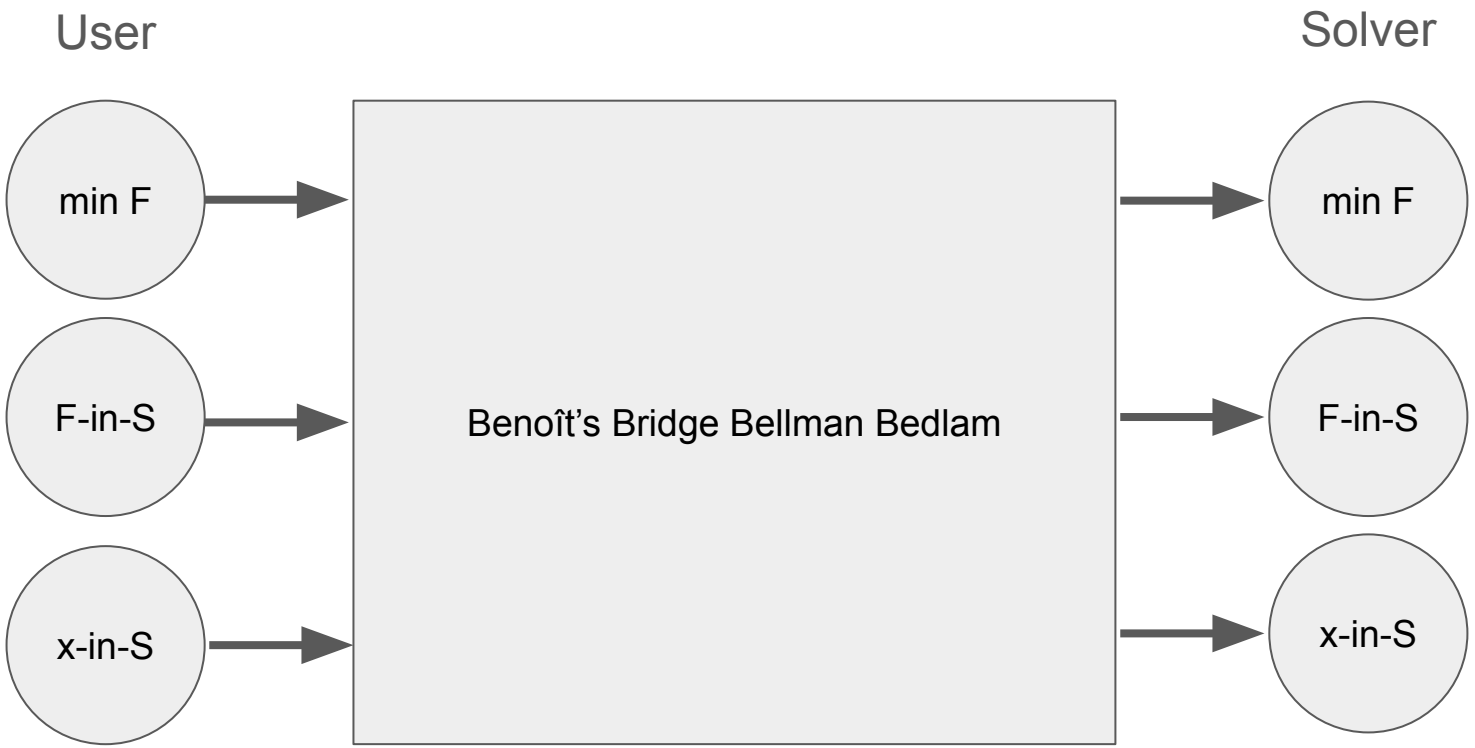


Solver

y in $(\mathbb{R}_+, \mathbb{R}_+)$

Hypergraphs and shortest hyperpaths

Complexity is hidden from the user



Hypergrapl

In practice, the

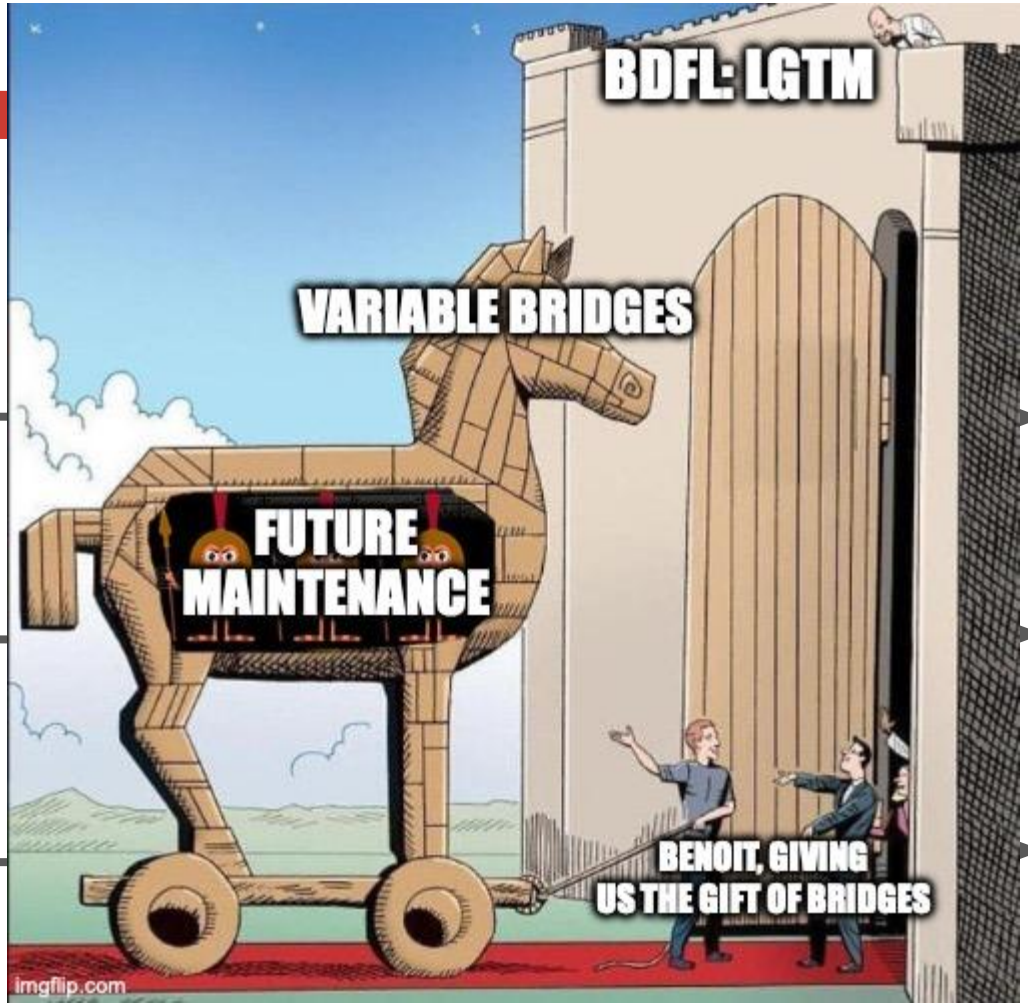
User

Solver

min F

F-in-S

x-in-S



min F

F-in-S

x-in-S

MathOptInterface.jl is a monolith

122,000 lines of code

/src	79,000 lines	/test	43,000 lines
/src/Bridges	19,800 lines	/test/Bridges	18,100 lines
/src/FileFormats	7,500 lines		
/src/Nonlinear	6,100 lines		
/src/Test	22,300 lines	/test/FileFormats	8,200 lines
		/test/Utilities	11,200 lines
/src/Utilities	14,800 lines		

MOI.

The top-level API

MOI uses multiple dispatch.

A lot.

Every solver is a new type.

Every function is a new type.

Every set is a new type.

There are a small number of public functions.

Testing takes a long time. Running the tests creates 6000 methods and 378,411 MethodInstances.

```
abstract type ModelLike end
abstract type AbstractOptimizer <: ModelLike end
abstract type AbstractFunction end
abstract type AbstractSet end

empty!(::ModelLike)
is_empty(::ModelLike)

add_variable(::ModelLike)::VariableIndex

add_constraint(
    ::ModelLike,
    ::AbstractFunction,
    ::AbstractSet,
)::ConstraintIndex

optimize!(::ModelLike)
```

MOI.

Functions

```
abstract type AbstractFunction <: MA.AbstractMutable end
abstract type AbstractScalarFunction <: AbstractFunction end
abstract type AbstractVectorFunction <: AbstractFunction end

struct VariableIndex <: AbstractScalarFunction
   value::Int64
end

struct ScalarAffineTerm{T}
   coefficient::T
   variable::VariableIndex
end

mutable struct ScalarAffineFunction{T} <: AbstractScalarFunction
   terms::Vector{ScalarAffineTerm{T}}
   constant::T
end
```

MOI.

Sets

```
abstract type AbstractSet end
abstract type AbstractScalarSet <: AbstractSet end
abstract type AbstractVectorSet <: AbstractSet end

struct EqualTo{T<:Number} <: AbstractScalarSet
   value::T
end

struct SecondOrderCone <: AbstractVectorSet
   dimension::Int
end

struct SOS1{T<:Real} <: AbstractVectorSet
   weights::Vector{T}
end

struct Indicator{A,S<:AbstractScalarSet} <: AbstractVectorSet
   set::S
end
```

MOI.Bridges

Benoît's Bridge Bellman Bedlam

This submodule contains:

- Three submodules for the different types of bridges: `Bridges.Constraint`, `Bridges.Objective`, and `Bridges.Variable`
- 76 bridges, each of which may cover many function-in-set combinations
- Bridges.LazyBridgeOptimizer, with Bellman-Ford and variable bridge substitutions
- Some other stuff, like tests for bridges

Users don't need to know the inner details

```
MOI.instantiate(
    HiGHS.Optimizer;
    with_bridge_type = Float64,
)
```

```
MOI.Bridges.full_bridge_optimizer(
    HiGHS.Optimizer(),
    Float64,
)
```

MOI.Bridges

The logic is horrific, so we try to test a lot

Testing is hard because of the number of possible permutations of supported variables, constraints, and objectives.

We test:

- every bridge in isolation
- a very wide range of unit tests
- every solver before every release, and sometimes with every PR

Sometimes this is still not enough. There are many edges cases.

Each bridge runs a common suite of tests

```
MOI.Bridges.runtests(
    MOI.Bridges.Constraint.ZeroOneBridge,
    """
    variables: x
    x in ZeroOne()
    """,
    """
    variables: x
    x in Integer()
    1.0 * x in Interval(0.0, 1.0)
    """,
)
```

```
Test Summary:     | Pass  Total  Time
Bridges.runtests  |   32     32  0.0s
```

MOI.FileFormats

Read and write models to disk

This submodule contains six submodules:

- CBF: the `.cbf` format
- LP: the `.lp` format
- MOF: the `.mof.json` format
- MPS: the `.mps` format
- NL: the `.nl` format
- SDPA: the `.sdpa` format

Each submodule implements

- A new `Model <: MOI.ModelLike`
- `Base.read!(io::IO, ::Model)`
- `Base.write(io::IO, ::Model)`

Downside to this design

To read and write models from disk, JuMP does a variation of:

```
function JuMP.write_to_file(
    model::Model, filename::String)
    dest = MOI.FileFormats.Model(; filename)
    MOI.copy_to(dest, model)
    MOI.write_to_file(dest, filename)
    return
end
```

We first have to create a copy of the entire model. This is slower than writing from the existing `model` object

MOI.FileFormats.MathOptFormat

https://jump.dev/MathOptFormat/

```
{
    "version": {"major": 1, "minor": 4},
    "variables": [{"name": "x"}],
    "objective": {
        "sense": "min",
        "function": {
            "type": "ScalarAffineFunction",
            "terms": [{"coefficient": 2, "variable": "x"}],
            "constant": 1
        }
    },
    "constraints": [{
        "function": {"type": "Variable", "name": "x"},
        "set": {"type": "GreaterThan", "lower": 1}
    }]
}
```

MathOptFormat supports every model you can write in MOI. And it has a schema.

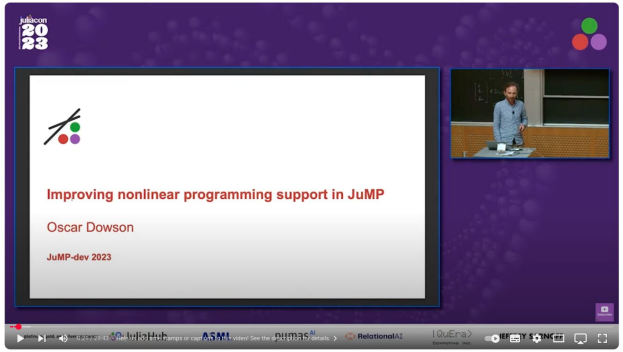
MOI.Nonlinear

A lot of complicated stuff

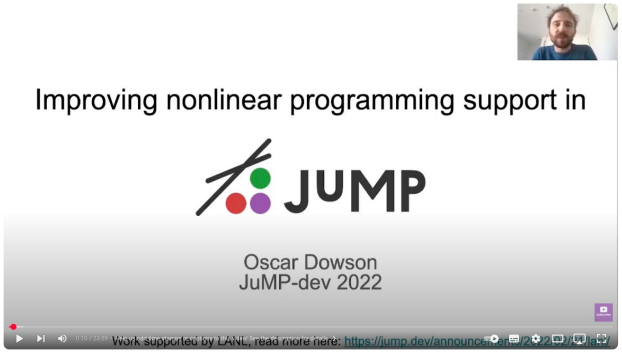
This submodule contains:

- Nonlinear.Model
  - A nonlinear modeling interface
- ReverseAD
  - A library for sparse reverse-mode automatic differentiation
- SymbolicAD
  - A symbolic differentiation library

See my JuMP-dev 2022 and 2023 talks.



Improving Nonlinear Programming Support in JuMP | Oscar Dowson | JuliaCon 2023



Improving Nonlinear Programming Support in JuMP | Oscar Dowson | JuliaCon 2022

MOI.Test

Test-Driven-Development of solver wrappers

This submodule contains

- A suite of >500 solver-independent test functions
- Accessible via `MOI.Test.runtests`
- Ability to include/exclude tests, adjust tolerances, etc

Downsides

- This (c|sh)ould have been a separate package
- Calling it `Test` was a mistake because it conflicts with Base.Test

Solvers get access to thousands of tests

```
using Test, HiGHS
import MathOptInterface as MOI
@testset "runtests" begin
    MOI.Test.runtests(
        MOI.instantiate(
            HiGHS.Optimizer;
            with_bridge_type = Float64,
        ),
        MOI.Test.Config(; atol = 1e-7),
    )
end
```

```
Test Summary: | Pass  Total     Time
runtests      | 3237   3237  4m23.8s
```

MOI.Utilities

A useful dumping ground for random stuff

This submodule contains a *loooot* of stuff

- Utilities for working with functions: comparing them, creating them, modifying them
- **Utilities.MockOptimizer**: a fake optimizer to mock tests in MOI and JuMP
- **Utilities.CachingOptimizer**: an abstraction across how solvers handle incremental modification
- **Utillities.Model** and **Utilities.UniversalFalllback**: model objects that support everything except optimize!
- **Utilities.CleverDicts**: a dict that starts as a Base.Vector but switches to a Base.Dict on deletion
- **Utilities.GenericModel**: a modular system for defining matrix-based storage

Just sooooo much other stuff…

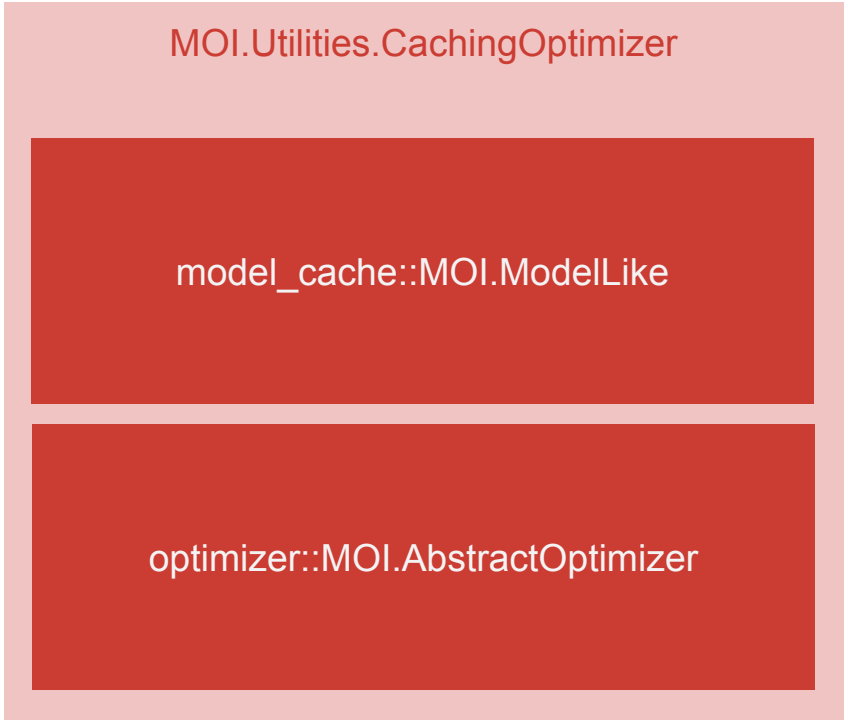
MOI.Utilities.CachingOptimizer

Abstract over differences in incremental modification

MOI.Utilities.CachingOptimizer:

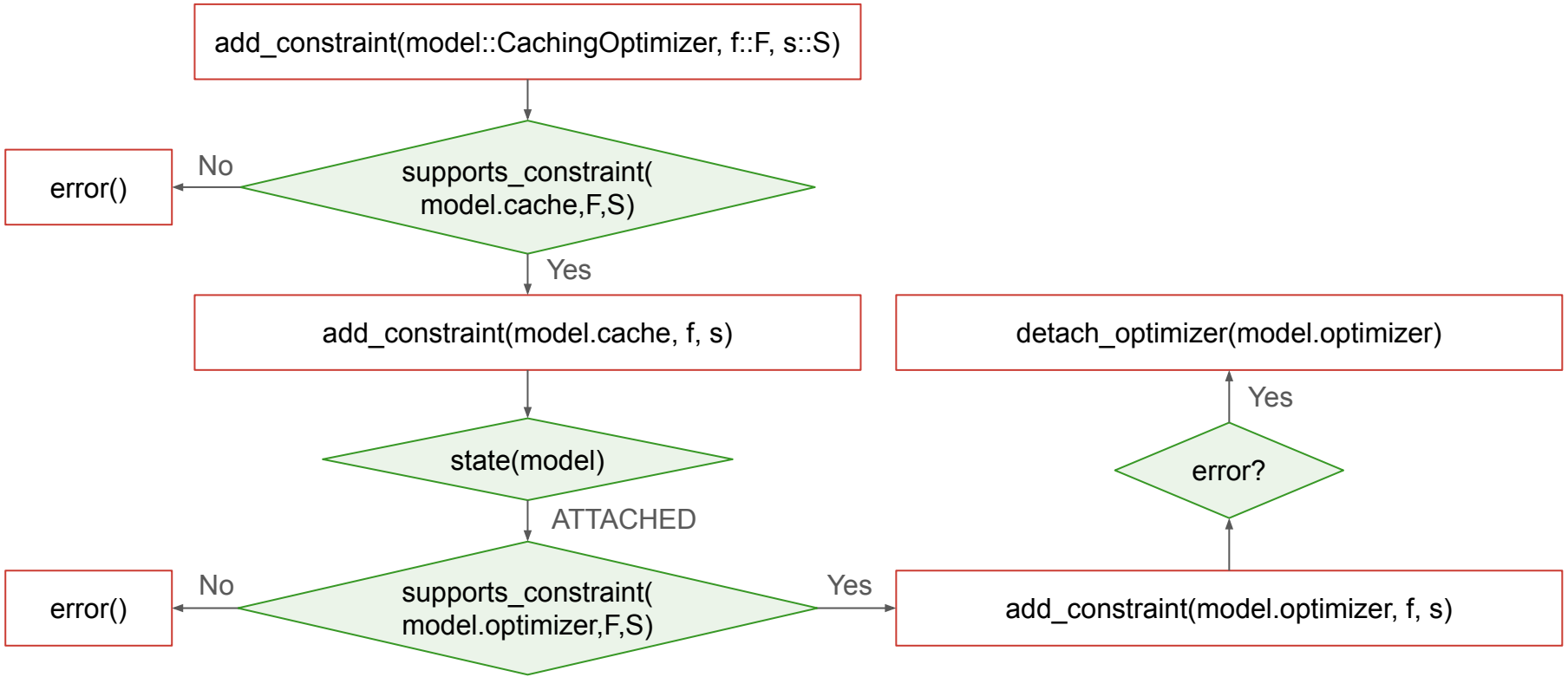
- maintains two copies of the model:
  - model_cache
  - optimizer

- is in one of three states
  - NO_OPTIMIZER
  - EMPTY_OPTIMIZER
  - ATTACHED_OPTIMIZER



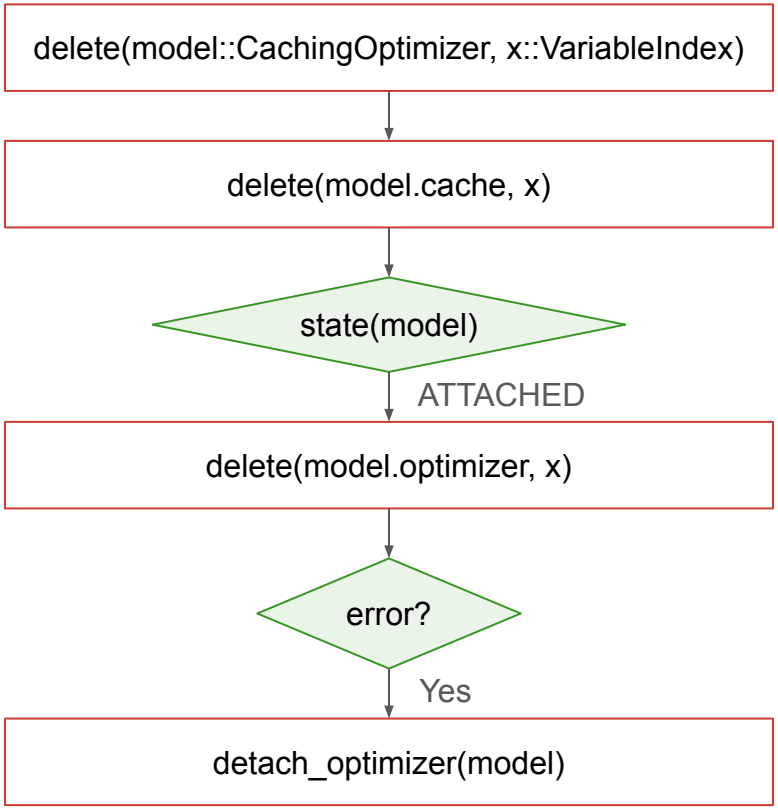
MOI.Utilities.CachingOptimizer

Flow for adding a constraint



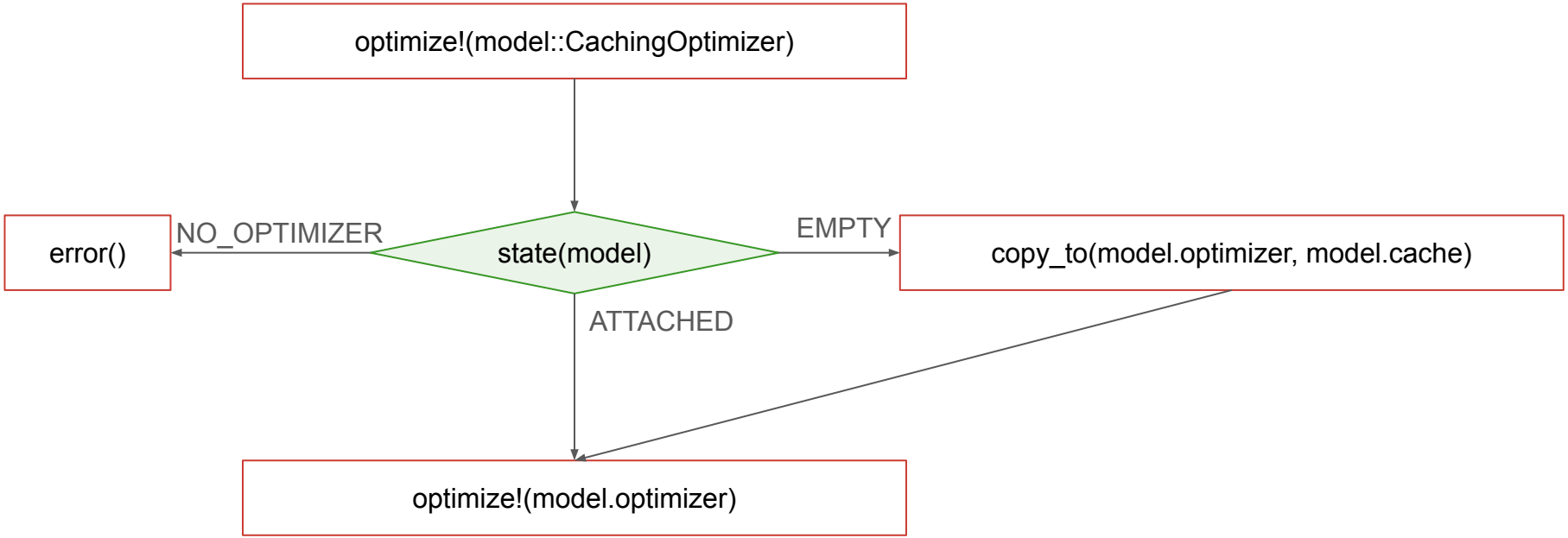
MOI.Utilities.CachingOptimizer

Flow for deleting a variable



MOI.Utilities.CachingOptimizer

Flow for calling optimize!



2013-2019: Problems with MathProgBase

There were many. Here are four.

Standard forms are not standard

- ECOS and SCS use different orderings for the exponential cone
- Gurobi does not support Ax in Interval
- CSDP does not support free variables

Some solvers mix problem classes

- KNITRO supported nonlinear, but also SecondOrderCone and Complements
- Gurobi was linear quadratic, but now also supports nonlinear

Extending the classes is hard

- No indicator constraints
- No complementarity constraints

In-place problem modification was limited

- No support for deleting variables or constraints
- Support for modifying RHS but not LHS

Problems with MathOptInterface

There are many. Here are four.

Arbitrary Indices

Each variable and constraint has an associated value::Int64. These do not need to be ordered and do not need to be contiguous. Constraints of different types can have the same value. Make them ordered by creating and unique by variable/constraint.

AbstractVectorFunction

Why didn't we just make it Base.Vector{<:AbstractScalarFunction}?

The 0.5 in ScalarQuadraticFunction

It's so hard to remember whether to * or / the 0.5. Just make the function a list of terms. Not a Q matrix.

Variable bridges (and constrained variables)

We haven't talked about `add_constrained_variable`. Variable bridges are hard because variable sets can overlap: `@variable(model, x >= 0, Int)` Is the domain of x Real, Nonnegatives, or Integer?

These problems are too breaking for us to ever consider changing. There will not be a MathOptInterface 2.0. We don't want Python 2->3.
But if you're looking to re-implement MOI in a different language… don't copy it blindly. Come talk to me.

The purpose of this talk

Why is this talk needed:

- MathOptInterface.jl (MOI) is one of the largest packages in all of Julia
- It is the connection between JuMP and solvers
- It uses a novel abstraction
- It has a looooooot of stuff in it

We haven't publicly talked about it much

By the end of this talk you will:

1. Understand the problem we are trying to solve and why we wrote MOI
2. Understand the MOI abstraction
3. Understand what a bridge is and why they are necessary
4. Have an overview of the components in MathOptInterface.jl

By the end of this talk you will not:

- Be able to write a solver wrapper
- Know how to write code that uses MOI