MathOptInterface

The JuMP core developers and contributors

January 25, 2022

Contents

Contents **ii**

I Introduction **1**

1 Introduction **2**
- 1.1 What is MathOptInterface? . . . 2
- 1.2 How the documentation is structured . . . 2
- 1.3 Citing MathOptInterface . . . 3

2 Motivation **4**

II Tutorials **5**

3 Solving a problem using MathOptInterface **6**
- 3.1 Required packages . . . 6
- 3.2 Define the data . . . 6
- 3.3 Add the variables . . . 7
- 3.4 Set the objective . . . 7
- 3.5 Add the constraints . . . 7
- 3.6 Optimize the model . . . 7
- 3.7 Understand why the solver stopped . . . 7
- 3.8 Understand what solution was returned . . . 8
- 3.9 Query the objective . . . 8
- 3.10 Query the primal solution . . . 8

4 Implementing a solver interface **9**
- 4.1 A note on the API . . . 9
- 4.2 Preliminaries . . . 9
- 4.3 Structuring the package . . . 11
- 4.4 Initial code . . . 12
- 4.5 The big decision: copy-to or incremental modifications? . . . 15
- 4.6 Names . . . 18
- 4.7 Solutions . . . 19
- 4.8 Other tips . . . 20

5 Transitioning from MathProgBase **22**
- 5.1 Transitioning a solver interface . . . 22
- 5.2 Transitioning the high-level functions . . . 22

6 Implementing a constraint bridge **24**
6.1 Preliminaries . . . 24
6.2 Four mandatory parts in a constraint bridge . . . 24
6.3 Bridge registration . . . 27
6.4 Bridge improvements . . . 27

7 Manipulating expressions **30**
7.1 Creating functions . . . 30
7.2 Canonicalizing functions . . . 31
7.3 Exploring functions . . . 32

8 Latency **33**
8.1 Background . . . 33
8.2 Causes . . . 33
8.3 Resolutions . . . 36

III Manual **39**

9 Standard form problem **40**
9.1 Functions . . . 40
9.2 One-dimensional sets . . . 41
9.3 Vector cones . . . 41
9.4 Matrix cones . . . 42
9.5 Multi-dimensional sets with combinatorial structure . . . 42

10 Models **43**
10.1 Attributes . . . 43
10.2 ModelLike API . . . 44
10.3 AbstractOptimizer API . . . 44

11 Variables **46**
11.1 Add a variable . . . 46
11.2 Delete a variable . . . 46
11.3 Variable attributes . . . 47

12 Constraints **48**
12.1 Add a constraint . . . 48
12.2 Delete a constraint . . . 49
12.3 Constraint attributes . . . 49
12.4 Constraints by function-set pairs . . . 49
12.5 JuMP mapping . . . 50

13 Solutions **52**
13.1 Solving and retrieving the results . . . 52
13.2 Why did the solver stop? . . . 52
13.3 Primal solutions . . . 52
13.4 Dual solutions . . . 53
13.5 Common status situations . . . 53

14 Problem modification **55**
14.1 Modify the set of a constraint . . . 55
14.2 Modify the function of a constraint . . . 56

14.3 Modify constant term in a scalar function . . . 57
14.4 Modify constant terms in a vector function . . . 57
14.5 Modify affine coefficients in a scalar function . . . 58
14.6 Modify affine coefficients in a vector function . . . 58

IV Background **60**

15 Duality **61**
15.1 Duality and scalar product . . . 62
15.2 Dual for problems with quadratic functions . . . 63

16 Infeasibility certificates **65**
16.1 Conic duality . . . 65
16.2 Unbounded problems . . . 66
16.3 Infeasible problems . . . 67

17 Naming conventions **68**
17.1 Sets . . . 68

V API Reference **69**

18 Standard form **70**
18.1 Functions . . . 70
18.2 Sets . . . 73
18.3 Scalar sets . . . 75
18.4 Vector sets . . . 77
18.5 Matrix sets . . . 81

19 Models **86**
19.1 Attribute interface . . . 86
19.2 Model interface . . . 90
19.3 Model attributes . . . 92
19.4 Optimizer interface . . . 94
19.5 Optimizer attributes . . . 95

20 Variables **103**
20.1 Functions . . . 103
20.2 Attributes . . . 106

21 Constraints **108**
21.1 Types . . . 108
21.2 Functions . . . 108
21.3 Attributes . . . 110

22 Modifications **114**

23 Nonlinear programming **116**
23.1 Types . . . 116
23.2 Attributes . . . 117
23.3 Functions . . . 117

24 Callbacks **120**
24.1 Attributes . . . 120
24.2 Lazy constraints . . . 121
24.3 User cuts . . . 122
24.4 Heuristic solutions . . . 123

25 Errors **125**

VI Submodules **130**

26 Benchmarks **131**
26.1 Overview . . . 131
26.2 API Reference . . . 132

27 Bridges **134**
27.1 Overview . . . 134
27.2 Implementation . . . 137
27.3 API Reference . . . 141

28 FileFormats **161**
28.1 Overview . . . 161
28.2 API Reference . . . 165

29 Utilities **167**
29.1 Overview . . . 167
29.2 API Reference . . . 175

30 Test **199**
30.1 Overview . . . 199
30.2 API Reference . . . 204

Part I

Introduction

Chapter 1

Introduction

Welcome to the documentation for MathOptInterface.

Note

This documentation is also available in PDF format: MathOptInterface.pdf.

1.1 What is MathOptInterface?

MathOptInterface.jl (MOI) is an abstraction layer designed to provide a unified interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs.

Tip

This documentation is aimed at developers writing software interfaces to solvers and modeling languages using the MathOptInterface API. If you are a user interested in solving optimization problems, we encourage you instead to use MOI through a higher-level modeling interface like JuMP or Convex.jl.

1.2 How the documentation is structured

Having a high-level overview of how this documentation is structured will help you know where to look for certain things.

- The **Tutorials** section contains articles on how to use and implement the MathOptInteraface API. Look here if you want to write a model in MOI, or write an interface to a new solver.
- The **Manual** contains short code-snippets that explain how to use the MOI API. Look here for more details on particular areas of MOI.
- The **Background** section contains articles on the theory behind MathOptInterface. Look here if you want to understand why, rather than how.
- The **API Reference** contains a complete list of functions and types that comprise the MOI API. Look here is you want to know how to use (or implement) a particular function.
- The **Submodules** section contains stand-alone documentation for each of the submodules within MOI. These submodules are not required to interface a solver with MOI, but they make the job much easier.

1.3 Citing MathOptInterface

A paper describing the design and features of MathOptInterface is available on arXiv.

If you find MathOptInterface useful in your work, we kindly request that you cite the following paper:

```
@article{legat2021mathoptinterface,
    title={{MathOptInterface}: a data structure for mathematical optimization problems},
    author={Legat, Beno{\^\i}t and Dowson, Oscar and Garcia, Joaquim Dias and Lubin, Miles},
    journal={INFORMS Journal on Computing},
    year={2021},
    doi={10.1287/ijoc.2021.1067},
    publisher={INFORMS}
}
```

Chapter 2

Motivation

MathOptInterface (MOI) is a replacement for MathProgBase, the first-generation abstraction layer for mathematical optimization previously used by JuMP and Convex.jl.

To address a number of limitations of MathProgBase, MOI is designed to:

- Be simple and extensible
  - unifying linear, quadratic, and conic optimization,
  - seamlessly facilitating extensions to essentially arbitrary constraints and functions (e.g., indicator constraints, complementarity constraints, and piecewise-linear functions)
- Be fast
  - by allowing access to a solver's in-memory representation of a problem without writing intermediate files (when possible)
  - by using multiple dispatch and avoiding requiring containers of nonconcrete types
- Allow a solver to return multiple results (e.g., a pool of solutions)
- Allow a solver to return extra arbitrary information via attributes (e.g., variable- and constraint-wise membership in an irreducible inconsistent subset for infeasibility analysis)
- Provide a greatly expanded set of status codes explaining what happened during the optimization procedure
- Enable a solver to more precisely specify which problem classes it supports
- Enable both primal and dual warm starts
- Enable adding and removing both variables and constraints by indices that are not required to be consecutive
- Enable any modification that the solver supports to an existing model
- Avoid requiring the solver wrapper to store an additional copy of the problem data

Part II

Tutorials

Chapter 3

Solving a problem using MathOptInterface

In this tutorial we demonstrate how to use MathOptInterface to solve the binary-constrained knapsack problem:

$$\begin{aligned} \max\ & c^\top x \\ s.t.\ & w^\top x \le C \\ & x_i \in \{0,1\}, \quad \forall i = 1,\dots,n \end{aligned}$$

3.1 Required packages

Load the MathOptInterface module and define the shorthand `MOI`:

```
using MathOptInterface
const MOI = MathOptInterface
```

As an optimizer, we choose GLPK:

```
using GLPK
optimizer = GLPK.Optimizer()
```

3.2 Define the data

We first define the constants of the problem:

```
julia> c = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> w = [0.3, 0.5, 1.0]
3-element Vector{Float64}:
 0.3
 0.5
 1.0

julia> C = 3.2
3.2
```

3.3 Add the variables

```
julia> x = MOI.add_variables(optimizer, length(c));
```

3.4 Set the objective

```
julia> MOI.set(
           optimizer,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(c, x), 0.0),
       );

julia> MOI.set(optimizer, MOI.ObjectiveSense(), MOI.MAX_SENSE)
```

Tip

`MOI.ScalarAffineTerm.(c, x)` is a shortcut for `[MOI.ScalarAffineTerm(c[i], x[i]) for i = 1:3]`. This is Julia's broadcast syntax in action, and is used quite often throughout MOI.

3.5 Add the constraints

We add the knapsack constraint and integrality constraints:

```
julia> MOI.add_constraint(
           optimizer,
           MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(w, x), 0.0),
           MOI.LessThan(C),
       );
```

Add integrality constraints:

```
julia> for x_i in x
           MOI.add_constraint(optimizer, x_i, MOI.ZeroOne())
       end
```

3.6 Optimize the model

```
julia> MOI.optimize!(optimizer)
```

3.7 Understand why the solver stopped

The first thing to check after optimization is why the solver stopped, e.g., did it stop because of a time limit or did it stop because it found the optimal solution?

```
julia> MOI.get(optimizer, MOI.TerminationStatus())
OPTIMAL::TerminationStatusCode = 1
```

Looks like we found an optimal solution!

3.8 Understand what solution was returned

```
julia> MOI.get(optimizer, MOI.ResultCount())
1

julia> MOI.get(optimizer, MOI.PrimalStatus())
FEASIBLE_POINT::ResultStatusCode = 1

julia> MOI.get(optimizer, MOI.DualStatus())
NO_SOLUTION::ResultStatusCode = 0
```

3.9 Query the objective

What is its objective value?

```
julia> MOI.get(optimizer, MOI.ObjectiveValue())
6.0
```

3.10 Query the primal solution

And what is the value of the variables x?

```
julia> MOI.get(optimizer, MOI.VariablePrimal(), x)
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

Chapter 4

Implementing a solver interface

This guide outlines the basic steps to implement an interface to MathOptInterface for a new solver.

Danger

Implementing an interface to MathOptInterface for a new solver is a lot of work. Before starting, we recommend that you join the Developer chatroom and explain a little bit about the solver you are wrapping. If you have questions that are not answered by this guide, please ask them in the Developer chatroom so we can improve this guide!

4.1 A note on the API

The API of MathOptInterface is large and varied. In order to support the diversity of solvers and use-cases, we make heavy use of duck-typing. That is, solvers are not expected to implement the full API, nor is there a well-defined minimal subset of what must be implemented. Instead, you should implement the API as necessary in order to make the solver function as you require.

The main reason for using duck-typing is that solvers work in different ways and target different use-cases.

For example:

- Some solvers support incremental problem construction, support modification after a solve, and have native support for things like variable names.
- Other solvers are "one-shot" solvers that require all of the problem data to construct and solve the problem in a single function call. They do not support modification or things like variable names.
- Other "solvers" are not solvers at all, but things like file readers. These may only support functions like `read_from_file`, and may not even support the ability to add variables or constraints directly!
- Finally, some "solvers" are layers which take a problem as input, transform it according to some rules, and pass the transformed problem to an inner solver.

4.2 Preliminaries

Before starting on your wrapper, you should do some background research and make the solver accessible via Julia.

Decide if MathOptInterface is right for you

The first step in writing a wrapper is to decide whether implementing an interface is the right thing to do.

MathOptInterface is an abstraction layer for unifying constrained mathematical optimization solvers. If your solver doesn't fit in the category, i.e., it implements a derivative-free algorithm for unconstrained objective functions, MathOptInterface may not be the right tool for the job.

> **Tip**
>
> If you're not sure whether you should write an interface, ask in the Developer chatroom.

Find a similar solver already wrapped

The next step is to find (if possible) a similar solver that is already wrapped. Although not strictly necessary, this will be a good place to look for inspiration when implementing your wrapper.

The JuMP documentation has a good list of solvers, along with the problem classes they support.

> **Tip**
>
> If you're not sure which solver is most similar, ask in the Developer chatroom.

Create a low-level interface

Before writing a MathOptInterface wrapper, you first need to be able to call the solver from Julia.

Wrapping solvers written in Julia

If your solver is written in Julia, there's nothing to do here! Go to the next section.

Wrapping solvers written in C

Julia is well suited to wrapping solvers written in C.

> **Info**
>
> This is not true for C++. If you have a solver written in C++, first write a C interface, then wrap the C interface.

Before writing a MathOptInterface wrapper, there are a few extra steps.

Create a JLL If the C code is publicly available under an open-source license, create a JLL package via Yggdrasil. The easiest way to do this is to copy an existing solver. Good examples to follow are the COIN-OR solvers.

> **Warning**
>
> Building the solver via Yggdrasil is non-trivial. Please ask the Developer chatroom for help.

If the code is commercial or not publicly available, the user will need to manually install the solver. See Gurobi.jl or CPLEX.jl for examples of how to structure this.

Use Clang.jl to wrap the C API The next step is to use Clang.jl to automatically wrap the C API. The easiest way to do this is to follow an example. Good examples to follow are Cbc.jl and HiGHS.jl.

Sometimes, you will need to make manual modifications to the resulting files.

Solvers written in other languages

Ask the Developer chatroom for advice. You may be able to use one of the JuliaInterop packages to call out to the solver.

For example, SeDuMi.jl uses MATLAB.jl to call the SeDuMi solver written in MATLAB.

4.3 Structuring the package

Structure your wrapper as a Julia package. Consult the Julia documentation if you haven't done this before.

MOI solver interfaces may be in the same package as the solver itself (either the C wrapper if the solver is accessible through C, or the Julia code if the solver is written in Julia, for example), or in a separate package which depends on the solver package.

> **Note**
>
> The JuMP core contributors request that you do not use "JuMP" in the name of your package without prior consent.

Your package should have the following structure:

```
/.github
    /workflows
        ci.yml
        format_check.yml
        TagBot.yml
/gen
    gen.jl  # Code to wrap the C API
/src
    NewSolver.jl
    /gen
        libnewsolver_api.jl
        libnewsolver_common.jl
    /MOI_wrapper
        MOI_wrapper.jl
        other_files.jl
/test
    runtests.jl
    /MOI_wrapper
        MOI_wrapper.jl
.gitignore
.JuliaFormatter.toml
README.md
LICENSE.md
Project.toml
```

- The `/.github` folder contains the scripts for GitHub actions. The easiest way to write these is to copy the ones from an existing solver.
- The `/gen` and `/src/gen` folders are only needed if you are wrapping a solver written in C.

- The `/src/MOI_wrapper` folder contains the Julia code for the MOI wrapper.
- The `/test` folder contains code for testing your package. See Setup tests for more information.
- The `.JuliaFormatter.toml` and `.github/workflows/format_check.yml` enforce code formatting using JuliaFormatter.jl. Check existing solvers or JuMP.jl for details.

Documentation

Your package must include documentation explaining how to use the package. The easiest approach is to include documentation in your `README.md`. A more involved option is to use Documenter.jl.

Examples of packages with README-based documentation include:

- Cbc.jl
- HiGHS.jl
- SCS.jl

Examples of packages with Documenter-based documentation include:

- Alpine.jl
- COSMO.jl
- Juniper.jl

Setup tests

The best way to implement an interface to MathOptInterface is via test-driven development.

The `MOI.Test` submodule contains a large test suite to help check that you have implemented things correctly.

Follow the guide How to test a solver to set up the tests for your package.

> **Tip**
>
> Run the tests frequently when developing. However, at the start there is going to be a lot of errors! Start by excluding large classes of tests (e.g., `exclude = ["test_basic_", "test_model_"]`, implement any missing methods until the tests pass, then remove an exclusion and repeat.

4.4 Initial code

By this point, you should have a package setup with tests, formatting, and access to the underlying solver. Now it's time to start writing the wrapper.

The `Optimizer` object

The first object to create is a subtype of `AbstractOptimizer`. This type is going to store everything related to the problem.

By convention, these optimizers should not be exported and should be named `PackageName.Optimizer`.

```
import MathOptInterface
const MOI = MathOptInterface

struct Optimizer <: MOI.AbstractOptimizer
    # Fields go here
end
```

Optimizer objects for C solvers

> **Warning**
>
> This section is important if you wrap a solver written in C.

Wrapping a solver written in C will require the use of pointers, and for you to manually free the solver's memory when the `Optimizer` is garbage collected by Julia.

Never pass a pointer directly to a Julia `ccall` function.

Instead, store the pointer as a field in your `Optimizer`, and implement `Base.cconvert` and `Base.unsafe_convert`. Then you can pass `Optimizer` to any `ccall` function that expects the pointer.

In addition, make sure you implement a `finalizer` for each model you create.

If `newsolver_createProblem()` is the low-level function that creates the problem pointer in C, and `newsolver_freeProblem(::Pt` is the low-level function that frees memory associated with the pointer, your `Optimizer()` function should look like this:

```
struct Optimizer <: MOI.AbstractOptimizer
    ptr::Ptr{Cvoid}

    function Optimizer()
        ptr = newsolver_createProblem()
        model = Optimizer(ptr)
        finalizer(model) do m
            newsolver_freeProblem(m)
            return
        end
        return model
    end
end

Base.cconvert(::Type{Ptr{Cvoid}}, model::Optimizer) = model
Base.unsafe_convert(::Type{Ptr{Cvoid}}, model::Optimizer) = model.ptr
```

Implement methods for `Optimizer`

All `Optimizers` must implement the following methods:

- `empty!`
- `is_empty`
- `optimize!`

Other methods, detailed below, are optional or depend on how you implement the interface.

Tip

For this and all future methods, read the docstrings to understand what each method does, what it expects as input, and what it produces as output. If it isn't clear, let us know and we will improve the docstrings! It is also very helpful to look at an existing wrapper for a similar solver.

You should also implement `Base.show(::IO, ::Optimizer)` to print a nice string when someone prints your model. For example

```
function Base.show(io::IO, model::Optimizer)
    return print(io, "NewSolver with the pointer $(model.ptr)")
end
```

Implement attributes

MathOptInterface uses attributes to manage different aspects of the problem.

For each attribute

- `get` gets the current value of the attribute
- `set` sets a new value of the attribute. Not all attributes can be set. For example, the user can't modify the `SolverName`.
- `supports` returns a `Bool` indicating whether the solver supports the attribute.

Info

Use `attribute_value_type` to check the value expected by a given attribute. You should make sure that your `get` function correctly infers to this type (or a subtype of it).

Each column in the table indicates whether you need to implement the particular method for each attribute.

Attribute	get	set	supports
SolverName	Yes	No	No
SolverVersion	Yes	No	No
RawSolver	Yes	No	No
Name	Yes	Yes	Yes
Silent	Yes	Yes	Yes
TimeLimitSec	Yes	Yes	Yes
RawOptimizerAttribute	Yes	Yes	Yes
NumberOfThreads	Yes	Yes	Yes

For example:

```
function MOI.get(model::Optimizer, ::MOI.Silent)
    return # true if MOI.Silent is set
end

function MOI.set(model::Optimizer, ::MOI.Silent, v::Bool)
    if v
        # Set a parameter to turn off printing
    else
```

```
        # Restore the default printing
    end
    return
end

MOI.supports(::Optimizer, ::MOI.Silent) = true
```

Define supports_constraint

The next step is to define which constraints and objective functions you plan to support.

For each function-set constraint pair, define supports_constraint:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
    ::Type{MOI.ZeroOne},
)
    return true
end
```

To make this easier, you may want to use Unions:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
    ::Type{<:Union{MOI.LessThan,MOI.GreaterThan,MOI.EqualTo}},
)
    return true
end
```

> **Tip**
>
> Only support a constraint if your solver has native support for it.

4.5 The big decision: copy-to or incremental modifications?

Now you need to decide whether to support incremental modification or not.

Incremental modification means that the user can add variables and constraints one-by-one without needing to rebuild the entire problem, and they can modify the problem data after an optimize! call. Supporting incremental modification means implementing functions like add_variable and add_constraint.

The alternative is to accept the problem data in a single copy_to function call, afterwhich it cannot be modified. Because copy_to sees all of the data at once, it can typically call a more efficient function to load data into the underlying solver.

Good examples of solvers supporting incremental modification are MILP solvers like GLPK.jl and Gurobi.jl. Examples of copy_to solvers are AmplNLWriter.jl and SCS.jl

It is possible to implement both approaches, but you should probably start with one for simplicity.

> **Tip**
>
> Only support incremental modification if your solver has native support for it.

In general, supporting incremental modification is more work, and it usually requires some extra book-keeping. However, it provides a more efficient interface to the solver if the problem is going to be resolved multiple times with small modifications. Moreover, once you've implemented incremental modification, it's usually not much extra work to add a `copy_to` interface. The converse is not true.

> **Tip**
>
> If this is your first time writing an interface, start with `copy_to`.

The `copy_to` interface

To implement the `copy_to` interface, implement the following function:

- `copy_to`

The incremental interface

> **Warning**
>
> Writing this interface is a lot of work. The easiest way is to consult the source code of a similar solver!

To implement the incremental interface, implement the following functions:

- `add_variable`
- `add_variables`
- `add_constraint`
- `add_constraints`
- `is_valid`
- `delete`

> **Info**
>
> Solvers do not have to support `AbstractScalarFunction` in `GreaterThan`, `LessThan`, `EqualTo`, or `Interval` with a nonzero constant in the function. Throw `ScalarFunctionConstantNotZero` if the function constant is not zero.

In addition, you should implement the following model attributes:

Attribute	`get`	`set`	`supports`
`ListOfModelAttributesSet`	Yes	No	No
`ObjectiveFunctionType`	Yes	No	No
`ObjectiveFunction`	Yes	Yes	Yes
`ObjectiveSense`	Yes	Yes	Yes
`Name`	Yes	Yes	Yes

Variable-related attributes:

Constraint-related attributes:

Attribute	get	set	supports
ListOfVariableAttributesSet	Yes	No	No
NumberOfVariables	Yes	No	No
ListOfVariableIndices	Yes	No	No

Attribute	get	set	supports
ListOfConstraintAttributesSet	Yes	No	No
NumberOfConstraints	Yes	No	No
ListOfConstraintTypesPresent	Yes	No	No
ConstraintFunction	Yes	Yes	No
ConstraintSet	Yes	Yes	No

Modifications

If your solver supports modifying data in-place, implement `modify` for the following `AbstractModifications`:

- `ScalarConstantChange`
- `ScalarCoefficientChange`
- `VectorConstantChange`
- `MultirowChange`

Variables constrained on creation

Some solvers require variables be associated with a set when they are created. This conflicts with the incremental modification approach, since you cannot first add a free variable and then constrain it to the set.

If this is the case, implement:

- `add_constrained_variable`
- `add_constrained_variables`
- `supports_add_constrained_variables`

By default, MathOptInterface assumes solvers support free variables. If your solver does not support free variables, define:

```
MOI.supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false
```

Incremental and copy_to

If you implement the incremental interface, you have the option of also implementing `copy_to`.

If you don't want to implement `copy_to`, e.g., because the solver has no API for building the problem in a single function call, define the following fallback:

```
MOI.supports_incremental_interface(::Optimizer) = true

function MOI.copy_to(dest::Optimizer, src::MOI.ModelLike)
    return MOI.Utilities.default_copy_to(dest, src)
end
```

4.6 Names

Regardless of which interface you implement, you have the option of implementing the Name attribute for variables and constraints:

Attribute	get	set	supports
VariableName	Yes	Yes	Yes
ConstraintName	Yes	Yes	Yes

If you implement names, you must also implement the following three methods:

```
function MOI.get(model::Optimizer, ::Type{MOI.VariableIndex}, name::String)
    return # The variable named `name`.
end

function MOI.get(model::Optimizer, ::Type{MOI.ConstraintIndex}, name::String)
    return # The constraint any type named `name`.
end

function MOI.get(
    model::Optimizer,
    ::Type{MOI.ConstraintIndex{F,S}},
    name::String,
) where {F,S}
    return # The constraint of type F-in-S named `name`.
end
```

These methods have the following rules:

- If there is no variable or constraint with the name, return nothing
- If there is a single variable or constraint with that name, return the variable or constraint
- If there are multiple variables or constraints with the name, throw an error.

Warning

You should not implement ConstraintName for VariableIndex constraints. If you implement ConstraintName for other constraints, you can add the following two methods to disable ConstraintName for VariableIndex constraints.

```
function MOI.supports(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::Type{<:MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet}},
)
    return throw(MOI.VariableIndexConstraintNameError())
end
function MOI.set(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet},
    ::String,
)
    return throw(MOI.VariableIndexConstraintNameError())
end
```

4.7 Solutions

Implement `optimize!` to solve the model:

- `optimize!`

All `Optimizers` must implement the following attributes:

- `DualStatus`
- `PrimalStatus`
- `RawStatusString`
- `ResultCount`
- `TerminationStatus`

Info

You only need to implement `get` for solution attributes. Don't implement `set` or `supports`.

Note

Solver wrappers should document how the low-level statuses map to the MOI statuses. Statuses like `NEARLY_FEASIBLE_POINT` and `INFEASIBLE_POINT`, are designed to be used when the solver explicitly indicates that relaxed tolerances are satisfied or the returned point is infeasible, respectively.

You should also implement the following attributes:

- `ObjectiveValue`
- `SolveTimeSec`
- `VariablePrimal`

Tip

Attributes like `VariablePrimal` and `ObjectiveValue` are indexed by the result count. Use `MOI.check_result_index_bounc` `attr)` to throw an error if the attribute is not available.

If your solver returns dual solutions, implement:

- `ConstraintDual`
- `DualObjectiveValue`

For integer solvers, implement:

- `ObjectiveBound`
- `RelativeGap`

If applicable, implement:

- `SimplexIterations`
- `BarrierIterations`
- `NodeCount`

If your solver uses the Simplex method, implement:

- `ConstraintBasisStatus`

If your solver accepts primal or dual warm-starts, implement:

- `VariablePrimalStart`
- `ConstraintDualStart`

4.8 Other tips

Here are some other points to be aware of when writing your wrapper.

Unsupported constraints at runtime

In some cases, your solver may support a particular type of constraint (e.g., quadratic constraints), but only if the data meets some condition (e.g., it is convex).

In this case, declare that you `support` the constraint, and throw `AddConstraintNotAllowed`.

Dealing with multiple variable bounds

MathOptInterface uses `VariableIndex` constraints to represent variable bounds. Defining multiple variable bounds on a single variable is not allowed.

Throw `LowerBoundAlreadySet` or `UpperBoundAlreadySet` if the user adds a constraint that results in multiple bounds.

Only throw if the constraints conflict. It is okay to add `VariableIndex`-in-`GreaterThan` and then `VariableIndex`-in-`LessThan`, but not `VariableIndex`-in-`Interval` and then `VariableIndex`-in-`LessThan`,

Expect duplicate coefficients

Solvers must expect that functions such as `ScalarAffineFunction` and `VectorQuadraticFunction` may contain duplicate coefficents.

For example, `ScalarAffineFunction([ScalarAffineTerm(x, 1), ScalarAffineTerm(x, 1)], 0.0)`.

Use `Utilities.canonical` to return a new function with the duplicate coefficients aggregated together.

Don't modify user-data

All data passed to the solver must be copied immediately to internal data structures. Solvers may not modify any input vectors and must assume that input vectors will not be modified by users in the future.

This applies, for example, to the `terms` vector in `ScalarAffineFunction`. Vectors returned to the user, e.g., via `ObjectiveFunction` or `ConstraintFunction` attributes, must not be modified by the solver afterwards. The in-place version of `get!` can be used by users to avoid extra copies in this case.

Column Generation

There is no special interface for column generation. If the solver has a special API for setting coefficients in existing constraints when adding a new variable, it is possible to queue modifications and new variables and then call the solver's API once all of the new coefficients are known.

Solver-specific attributes

You don't need to restrict yourself to the attributes defined in the MathOptInterface.jl package.

Solver-specific attributes should be specified by creating an appropriate subtype of `AbstractModelAttribute`, `AbstractOptimizerAttribute`, `AbstractVariableAttribute`, or `AbstractConstraintAttribute`.

For example, Gurobi.jl adds attributes for multiobjective optimization by defining:

```
struct NumberOfObjectives <: MOI.AbstractModelAttribute end

function MOI.set(model::Optimizer, ::NumberOfObjectives, n::Integer)
    # Code to set NumberOfObjectives
    return
end

function MOI.get(model::Optimizer, ::NumberOfObjectives)
    n = # Code to get NumberOfObjectives
    return n
end
```

Then, the user can write:

```
model = Gurobi.Optimizer()
MOI.set(model, Gurobi.NumberofObjectives(), 3)
```

Chapter 5

Transitioning from MathProgBase

MathOptInterface is a replacement for MathProgBase.jl. However, it is not a direct replacement.

5.1 Transitioning a solver interface

MathOptInterface is more extensive than MathProgBase which may make its implementation seem daunting at first. There are however numerous utilities in MathOptInterface that the simplify implementation process.

For more information, read Implementing a solver interface.

5.2 Transitioning the high-level functions

MathOptInterface doesn't provide replacements for the high-level interfaces in MathProgBase. We recommend you use JuMP as a modeling interface instead.

> **Tip**
>
> If you haven't used JuMP before, start with the tutorial Getting started with JuMP

linprog

Here is one way of transitioning from `linprog`:

```
using JuMP

function linprog(c, A, sense, b, l, u, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, l[i] <= x[i=1:N] <= u[i])
    @objective(model, Min, c' * x)
    eq_rows, ge_rows, le_rows = sense .== '=', sense .== '>', sense .== '<'
    @constraint(model, A[eq_rows, :] * x .== b[eq_rows])
    @constraint(model, A[ge_rows, :] * x .>= b[ge_rows])
    @constraint(model, A[le_rows, :] * x .<= b[le_rows])
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

mixintprog

Here is one way of transitioning from `mixintprog`:

```
using JuMP

function mixintprog(c, A, rowlb, rowub, vartypes, lb, ub, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, lb[i] <= x[i=1:N] <= ub[i])
    for i in 1:N
        if vartypes[i] == :Bin
            set_binary(x[i])
        elseif vartypes[i] == :Int
            set_integer(x[i])
        end
    end
    @objective(model, Min, c' * x)
    @constraint(model, rowlb .<= A * x .<= rowub)
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

quadprog

Here is one way of transitioning from `quadprog`:

```
using JuMP

function quadprog(c, Q, A, rowlb, rowub, lb, ub, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, lb[i] <= x[i=1:N] <= ub[i])
    @objective(model, Min, c' * x + 0.5 * x' * Q * x)
    @constraint(model, rowlb .<= A * x .<= rowub)
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

Chapter 6

Implementing a constraint bridge

This guide outlines the basic steps to create a new bridge from a constraint expressed in the formalism `Function-in-Set`.

6.1 Preliminaries

First, decide on the set you want to bridge. Then, study its properties: the most important one is whether the set is scalar or vector, which impacts the dimensionality of the functions that can be used with the set.

- A scalar function only has one dimension. MOI defines three types of scalar functions: a variable (`VariableIndex`), an affine function (`ScalarAffineFunction`), or a quadratic function (`ScalarQuadraticFunction`).
- A vector function has several dimensions (at least one). MOI defines three types of vector functions: several variables (`VectorOfVariables`), an affine function (`VectorAffineFunction`), or a quadratic function (`VectorQuadraticFunction`). The main difference with scalar functions is that the order of dimensions can be very important: for instance, in an indicator constraint (`Indicator`), the first dimension indicates whether the constraint about the second dimension is active.

To explain how to implement a bridge, we present the example of `Bridges.Constraint.FlipSignBridge`. This bridge maps <= (`LessThan`) constraints to >= (`GreaterThan`) constraints. This corresponds to reversing the sign of the inequality. We focus on scalar affine functions (we disregard the cases of a single variable or of quadratic functions). This example is a simplified version of the code included in MOI.

6.2 Four mandatory parts in a constraint bridge

The first part of a constraint bridge is a new concrete subtype of `Bridges.Constraint.AbstractBridge`. This type must have fields to store all the new variables and constraints that the bridge will add. Typically, these types are parametrized by the type of the coefficients in the model.

Then, three sets of functions must be defined:

1. `Bridges.Constraint.bridge_constraint`: this function implements the bridge and creates the required variables and constraints.
2. `supports_constraint`: these functions must return `true` when the combination of function and set is supported by the bridge. By default, the base implementation always returns `false` and the bridge does not have to provide this implementation.

3. `Bridges.added_constrained_variable_types` and `Bridges.added_constraint_types`: these functions return the types of variables and constraints that this bridge adds. They are used to compute the set of other bridges that are required to use the one you are defining, if need be.

More functions can be implemented, for instance to retrieve properties from the bridge or deleting a bridged constraint.

1. Structure for the bridge

A typical `struct` behind a bridge depends on the type of the coefficients that are used for the model (typically `Float64`, but coefficients might also be integers or complex numbers).

This structure must hold a reference to all the variables and the constraints that are created as part of the bridge.

The type of this structure is used throughout MOI as an identifier for the bridge. It is passed as argument to most functions related to bridges.

The best practice is to have the name of this type end with `Bridge`.

In our example, the bridge maps any `ScalarAffineFunction{T}`-in-`LessThan{T}` constraint to a single `ScalarAffineFunction{T`
in-`GreaterThan{T}` constraint. The affine function has coefficients of type T. The bridge is parametrized with T, so that the constraint that the bridge creates also has coefficients of type T.

```
struct SignBridge{T<:Number} <: Bridges.Constraint.AbstractBridge
    constraint::ConstraintIndex{ScalarAffineFunction{T}, GreaterThan{T}}
end
```

2. Bridge creation

The function `Bridges.Constraint.bridge_constraint` is called whenever the bridge is instantiated for a specific model, with the given function and set. The arguments to `bridge_constraint` are similar to `add_constraint`, with the exception of the first argument: it is the `Type` of the struct defined in the first step (for our example, `Type{SignBridge{T}}`).

`bridge_constraint` returns an instance of the struct defined in the first step. the first step.

In our example, the bridge constraint could be defined as:

```
function Bridges.Constraint.bridge_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    model::ModelLike, # Model to which the constraint is being added.
    f::ScalarAffineFunction{T}, # Function to rewrite.
    s::LessThan{T}, # Set to rewrite.
) where {T}
    # Create the variables and constraints required for the bridge.
    con = add_constraint(model, -f, GreaterThan(-s.upper))

    # Return an instance of the bridge type with a reference to all the
    # variables and constraints that were created in this function.
    return SignBridge(con)
end
```

3. Supported constraint types

The function supports_constraint determines whether the bridge type supports a given combination of function and set.

This function must closely match bridge_constraint, because it will not be called if supports_constraint returns false.

```
function supports_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    ::Type{ScalarAffineFunction{T}}, # Function to rewrite.
    ::Type{LessThan{T}}, # Set to rewrite.
) where {T}
    # Do some computation to ensure that the constraint is supported.
    # Typically, you can directly return true.
    return true
end
```

4. Metadata about the bridge

To determine whether a bridge can be used, MOI uses a shortest-path algorithm that uses the variable types and the constraints that the bridge can create. This information is communicated from the bridge to MOI using the functions Bridges.added_constrained_variable_types and Bridges.added_constraint_types. Both return lists of tuples: either a list of 1-tuples containing the variable types (typically, ZeroOne or Integer) or a list of 2-tuples contained the functions and sets (like ScalarAffineFunction{T}-GreaterThan).

For our example, the bridge does not create any constrained variables, and only ScalarAffineFunction{T}-in-GreaterThan{T} constraints:

```
function Bridges.added_constrained_variable_types(::Type{SignBridge{T}}) where {T}
    # The bridge does not create variables, return an empty list of tuples:
    return Tuple{Type}[]
end

function Bridges.added_constraint_types(::Type{SignBridge{T}}) where {T}
    return Tuple{Type,Type}[
        # One element per F-in-S the bridge creates.
        (ScalarAffineFunction{T}, GreaterThan{T}),
    ]
end
```

A bridge that creates binary variables would rather have this definition of added_constrained_variable_types:

```
function Bridges.added_constrained_variable_types(::Type{SomeBridge{T}}) where {T}
    # The bridge only creates binary variables:
    return Tuple{Type}[(ZeroOne,)]
end
```

> **Warning**
>
> If you declare the creation of constrained variables in added_constrained_variable_types, the corresponding constraint type VariableIndex must not be indicated in added_constraint_types. This would restrict the use of the bridge to solvers that can add such a constraint after the variable is created.

> More concretely, if you declare in `added_constrained_variable_types` that your bridge creates binary variables (`ZeroOne`), and if you never add such a constraint afterward (you do not call `add_constraint(model, var, ZeroOne())`), then you must not list (`VariableIndex, ZeroOne`) in `added_constraint_types`.

Typically, the function `Bridges.Constraint.concrete_bridge_type` does not have to be defined for most bridges.

6.3 Bridge registration

For a bridge to be used by MOI, it must be known by MOI.

`SingleBridgeOptimizer`

The first way to do so is to create a single-bridge optimizer. This type of optimizer wraps another optimizer and adds the possibility to use only one bridge. It is especially useful when unit testing bridges.

It is common practice to use the same name as the type defined for the bridge (`SignBridge`, in our example) without the suffix `Bridge`.

```
const Sign{T,OT<: ModelLike} =
    SingleBridgeOptimizer{SignBridge{T}, OT}
```

In the context of unit tests, this bridge is used in conjunction with a `Utilities.MockOptimizer`:

```
mock = Utilities.MockOptimizer(
    Utilities.UniversalFallback(Utilities.Model{Float64}()),
)
bridged_mock = Sign{Float64}(mock)
```

New bridge for a `LazyBridgeOptimizer`

Typical user-facing models for MOI are based on `Bridges.LazyBridgeOptimizer`. For instance, this type of model is returned by `Bridges.full_bridge_optimizer`. These models can be added more bridges by using `Bridges.add_bridge`:

```
inner_optimizer = Utilities.Model{Float64}()
optimizer = Bridges.full_bridge_optimizer(inner_optimizer, Float64)
Bridges.add_bridge(optimizer, SignBridge{Float64})
```

6.4 Bridge improvements

Attribute retrieval

Like models, bridges have attributes that can be retrieved using `get` and `set`. The most important ones are the number of variables and constraints, but also the lists of variables and constraints.

In our example, we only have one constraint and only have to implement the `NumberOfConstraints` and `ListOfConstraintIndices` attributes:

```
function get(
    ::SignBridge{T},
    ::NumberOfConstraints{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return 1
end

function get(
    bridge::SignBridge{T},
    ::ListOfConstraintIndices{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return [bridge.constraint]
end
```

You must implement one such pair of functions for each type of constraint the bridge adds to the model.

Warning

Avoid returning a list from the bridge object without copying it. Users must be able to change the contents of the returned list without altering the bridge object.

For variables, the situation is simpler. If your bridge creates new variables, you must implement the `NumberOfVariables` and `ListOfVariableIndices` attributes. However, these attributes do not have parameters, unlike their constraint counterparts. Only two functions suffice:

```
function get(
    ::SignBridge{T},
    ::NumberOfVariables,
) where {T}
    return 0
end

function get(
    ::SignBridge{T},
    ::ListOfVariableIndices,
) where {T}
    return VariableIndex[]
end
```

Model modifications

To avoid copying the model when the user request to change a constraint, MOI provides `modify`. Bridges can also implement this API to allow certain changes, such as coefficient changes.

In our case, a modification of a coefficient in the original constraint (i.e. replacing the value of the coefficient of a variable in the affine function) must be transmitted to the constraint created by the bridge, but with a sign change.

```julia
function modify(
    model::ModelLike,
    bridge::SignBridge,
    change::ScalarCoefficientChange,
)
    modify(
        model,
        bridge.constraint,
        ScalarCoefficientChange(change.variable, -change.new_coefficient),
    )
    return
end
```

Bridge deletion

When a bridge is deleted, the constraints it added must be deleted too.

```julia
function delete(model::ModelLike, bridge::SignBridge)
    delete(model, bridge.constraint)
    return
end
```

Chapter 7

Manipulating expressions

This guide highlights a syntactically appealing way to build expressions at the MOI level, but also to look at their contents. It may be especially useful when writing models or bridge code.

7.1 Creating functions

This section details the ways to create functions with MathOptInterface.

Creating scalar affine functions

The simplest scalar function is simply a variable:

```
julia> x = MOI.add_variable(model) # Create the variable x
MathOptInterface.VariableIndex(1)
```

This type of function is extremely simple; to express more complex functions, other types must be used. For instance, a `ScalarAffineFunction` is a sum of linear terms (a factor times a variable) and a constant. Such an object can be built using the standard constructor:

```
julia> f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1, x)], 2) # x + 2
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 2)
```

However, you can also use operators to build the same scalar function:

```
julia> f = x + 2
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 2)
```

Creating scalar quadratic functions

Scalar quadratic functions are stored in `ScalarQuadraticFunction` objects, in a way that is highly similar to scalar affine functions. You can obtain a quadratic function as a product of affine functions:

```
julia> 1 * x * x
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[], 0)
```

```
julia> f * f  # (x + 2)²
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)), MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1))], 4)

julia> f^2  # (x + 2)² too
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)), MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1))], 4)
```

Creating vector functions

A vector function is a function with several values, irrespective of the number of input variables. Similarly to scalar functions, there are three main types of vector functions: `VectorOfVariables`, `VectorAffineFunction`, and `VectorQuadraticFunction`.

The easiest way to create a vector function is to stack several scalar functions using `Utilities.vectorize`. It takes a vector as input, and the generated vector function (of the most appropriate type) has each dimension corresponding to a dimension of the vector.

```
julia> g = MOI.Utilities.vectorize([f, 2 * f])
MathOptInterface.VectorAffineFunction{Int64}(MathOptInterface.VectorAffineTerm{Int64}[MathOptInterface.VectorAffineTer
↪  MathOptInterface.ScalarAffineTerm{Int64}(1, MathOptInterface.VariableIndex(1))),
↪  MathOptInterface.VectorAffineTerm{Int64}(2, MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)))], [2, 4])
```

Warning

`Utilities.vectorize` only takes a vector of similar scalar functions: you cannot mix `VariableIndex` and `ScalarAffineFunction`, for instance. In practice, it means that `Utilities.vectorize([x, f])` does not work; you should rather use `Utilities.vectorize([1 * x, f])` instead to only have `ScalarAffineFunction` objects.

7.2 Canonicalizing functions

In more advanced use cases, you might need to ensure that a function is "canonical". Functions are stored as an array of terms, but there is no check that these terms are redundant: a `ScalarAffineFunction` object might have two terms with the same variable, like `x + x + 1`. These terms could be merged without changing the semantics of the function: `2x + 1`.

Working with these objects might be cumbersome. Canonicalization helps maintain redundancy to zero.

`Utilities.is_canonical` checks whether a function is already in its canonical form:

```
julia> MOI.Utilities.is_canonical(f + f) # (x + 2) + (x + 2) is stored as x + x + 4
false
```

`Utilities.canonical` returns the equivalent canonical version of the function:

```
julia> MOI.Utilities.canonical(f + f) # Returns 2x + 4
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 4)
```

7.3 Exploring functions

At some point, you might need to dig into a function, for instance to map it into solver constructs.

Vector functions

`Utilities.scalarize` returns a vector of scalar functions from a vector function:

```
julia> MOI.Utilities.scalarize(g) # Returns a vector [f, 2 * f].
2-element Vector{MathOptInterface.ScalarAffineFunction{Int64}}:

↪  MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffin
↪  MathOptInterface.VariableIndex(1))], 2)

↪  MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffin
↪  MathOptInterface.VariableIndex(1))], 4)
```

Note

`Utilities.eachscalar` returns an iterator on the dimensions, which serves the same purpose as `Utilities.scalarize`.

`output_dimension` returns the number of dimensions of the output of a function:

```
julia> MOI.output_dimension(g)
2
```

Chapter 8

Latency

MathOptInterface suffers the "time-to-first-solve" problem of start-up latency.

This hurts both the user- and developer-experience of MathOptInterface. In the first case, because simple models have a multi-second delay before solving, and in the latter, because our tests take so long to run!

This page contains some advice on profiling and fixing latency-related problems in the MathOptInterface.jl repository.

8.1 Background

Before reading this part of the documentation, you should familiarize yourself with the reasons for latency in Julia and how to fix them.

- Read the blogposts on julialang.org on precompilation and SnoopCompile
- Read the SnoopCompile documentation.
- Watch Tim Holy's talk at JuliaCon 2021
- Watch the package development workshop at JuliaCon 2021

8.2 Causes

There are three main causes of latency in MathOptInterface:

1. A large number of types
2. Lack of method ownership
3. Type-instability in the bridge layer

A large number of types

Julia is very good at specializing method calls based on the input type. Each specialization has a compilation cost, but the benefit of faster run-time performance.

The best-case scenario is for a method to be called a large number of times with a single set of argument types. The worst-case scenario is for a method to be called a single time for a large set of argument types.

Because of MathOptInterface's function-in-set formulation, we fall into the worst-case situation.

This is a fundamental limitation of Julia, so there isn't much we can do about it. However, if we can precompile MathOptInterface, much of the cost can be shifted from start-up latency to the time it takes to precompile a package on installation.

However, there are two things which make MathOptInterface hard to precompile...

Lack of method ownership

Lack of method ownership happens when a call is made using a mix of structs and methods from different modules. Because of this, no single module "owns" the method that is being dispatched, and so it cannot be precompiled.

> **Tip**
>
> This is a slightly simplified explanation. Read the precompilation tutorial for a more in-depth discussion on back-edges.

Unfortunately, the design of MOI means that this is a frequent occurrence! We have a bunch of types in `MOI.Utilities` that wrap types defined in external packages (i.e., the `Optimizers`), which implement methods of functions defined in `MOI` (e.g., `add_variable`, `add_constraint`).

Here's a simple example of method-ownership in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize!(x::Wrapper) = optimize!(x.inner)
end  # MyMOI

module MyOptimizer
using ..MyMOI
struct Optimizer end
MyMOI.optimize!(x::Optimizer) = 1
end  # MyOptimizer

using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())

julia> tinf = @snoopi_deep MyMOI.optimize!(model)
InferenceTimingNode: 0.008256/0.008543 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with 1
↪  direct children
```

The result is that there was one method that required type inference. If we visualize `tinf`:

```
using ProfileView
ProfileView.view(flamegraph(tinf))
```

we see a flamegraph with a large red-bar indicating that the method `MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer}` cannot be precompiled.

To fix this, we need to designate a module to "own" that method (i.e., create a back-edge). The easiest way to do this is for `MyOptimizer` to call `MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer})` during `using MyOptimizer`. Let's see that in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize(x::Wrapper) = optimize(x.inner)
end  # MyMOI

module MyOptimizer
using ..MyMOI
struct Optimizer end
MyMOI.optimize(x::Optimizer) = 1
# The syntax of this let-while loop is very particular:
#  * `let ... end` keeps everything local to avoid polluting the MyOptimizer
#    namespace
#  * `while true ... break end` runs the code once, and forces Julia to compile
#    the inner loop, rather than interpret it.
let
    while true
        model = MyMOI.Wrapper(Optimizer())
        MyMOI.optimize(model)
        break
    end
end
end  # MyOptimizer

using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())

julia> tinf = @snoopi_deep MyMOI.optimize(model)
InferenceTimingNode: 0.006822/0.006822 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with 0
↪  direct children
```

There are now `0` direct children that required type inference because the method was already stored in `MyOptimizer`!

Unfortunately, this trick only works if the call-chain is fully inferrable. If there are breaks (due to type instability), then the benefit of doing this is reduced. And unfortunately for us, the design of MathOptInterface has a lot of type instabilities...

Type instability in the bridge layer

Most of MathOptInterface is pretty good at ensuring type-stability. However, a key component is not type stable, and that is the bridging layer.

In particular, the bridging layer defines `Bridges.LazyBridgeOptimizer`, which has fields like:

```
struct LazyBridgeOptimizer
    constraint_bridge_types::Vector{Any}
    constraint_node::Dict{Tuple{Type,Type},ConstraintNode}
    constraint_types::Vector{Tuple{Type,Type}}
end
```

This is because the `LazyBridgeOptimizer` needs to be able to deal with any function-in-set type passed to it, and we also allow users to pass additional bridges that they defined in external packages.

So to recap, MathOptInterface suffers package latency because:

1. there are a large number of types and functions...
2. and these are split between multiple modules, including external packages...
3. and there are type-instabilities like those in the bridging layer.

8.3 Resolutions

There are no magic solutions to reduce latency. Issue #1313 tracks progress on reducing latency in MathOpt-Interface.

A useful script is the following (replace GLPK as needed):

```
using MathOptInterface, GLPK
const MOI = MathOptInterface

function example_diet(optimizer, bridge)
    category_data = [
        1800.0 2200.0;
          91.0    Inf;
           0.0   65.0;
           0.0 1779.0
    ]
    cost = [2.49, 2.89, 1.50, 1.89, 2.09, 1.99, 2.49, 0.89, 1.59]
    food_data = [
        410 24 26 730;
        420 32 10 1190;
        560 20 32 1800;
        380  4 19 270;
        320 12 10 930;
        320 15 12 820;
        320 31 12 1230;
        100  8 2.5 125;
        330  8 10 180
    ]
    bridge_model = if bridge
        MOI.instantiate(optimizer; with_bridge_type=Float64)
    else
        MOI.instantiate(optimizer)
    end
    model = MOI.Utilities.CachingOptimizer(
        MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}()),
        MOI.Utilities.AUTOMATIC,
    )
    MOI.Utilities.reset_optimizer(model, bridge_model)
    MOI.set(model, MOI.Silent(), true)
    nutrition = MOI.add_variables(model, size(category_data, 1))
    for (i, v) in enumerate(nutrition)
        MOI.add_constraint(model, v, MOI.GreaterThan(category_data[i, 1]))
        MOI.add_constraint(model, v, MOI.LessThan(category_data[i, 2]))
    end
    buy = MOI.add_variables(model, size(food_data, 1))
    MOI.add_constraint.(model, buy, MOI.GreaterThan(0.0))
    MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
    f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(cost, buy), 0.0)
    MOI.set(model, MOI.ObjectiveFunction{typeof(f)}(), f)
```

```
    for (j, n) in enumerate(nutrition)
        f = MOI.ScalarAffineFunction(
            MOI.ScalarAffineTerm.(food_data[:, j], buy),
            0.0,
        )
        push!(f.terms, MOI.ScalarAffineTerm(-1.0, n))
        MOI.add_constraint(model, f, MOI.EqualTo(0.0))
    end
    MOI.optimize!(model)
    term_status = MOI.get(model, MOI.TerminationStatus())
    @assert term_status == MOI.OPTIMAL
    MOI.add_constraint(
        model,
        MOI.ScalarAffineFunction(
            MOI.ScalarAffineTerm.(1.0, [buy[end-1], buy[end]]),
            0.0,
        ),
        MOI.LessThan(6.0),
    )
    MOI.optimize!(model)
    @assert MOI.get(model, MOI.TerminationStatus()) == MOI.INFEASIBLE
    return
end

if length(ARGS) > 0
    bridge = get(ARGS, 2, "") != "--no-bridge"
    println("Running: $(ARGS[1]) $(get(ARGS, 2, ""))")
    @time example_diet(GLPK.Optimizer, bridge)
    @time example_diet(GLPK.Optimizer, bridge)
    exit(0)
end
```

You can create a flame-graph via

```
using SnoopComile
tinf = @snoopi_deep example_diet(GLPK.Optimizer, true)
using ProfileView
ProfileView.view(flamegraph(tinf))
```

Here's how things looked in mid-August 2021:

There are a few opportunities for improvement (non-red flames, particularly on the right). But the main problem is a large red (non-precompilable due to method ownership) flame.

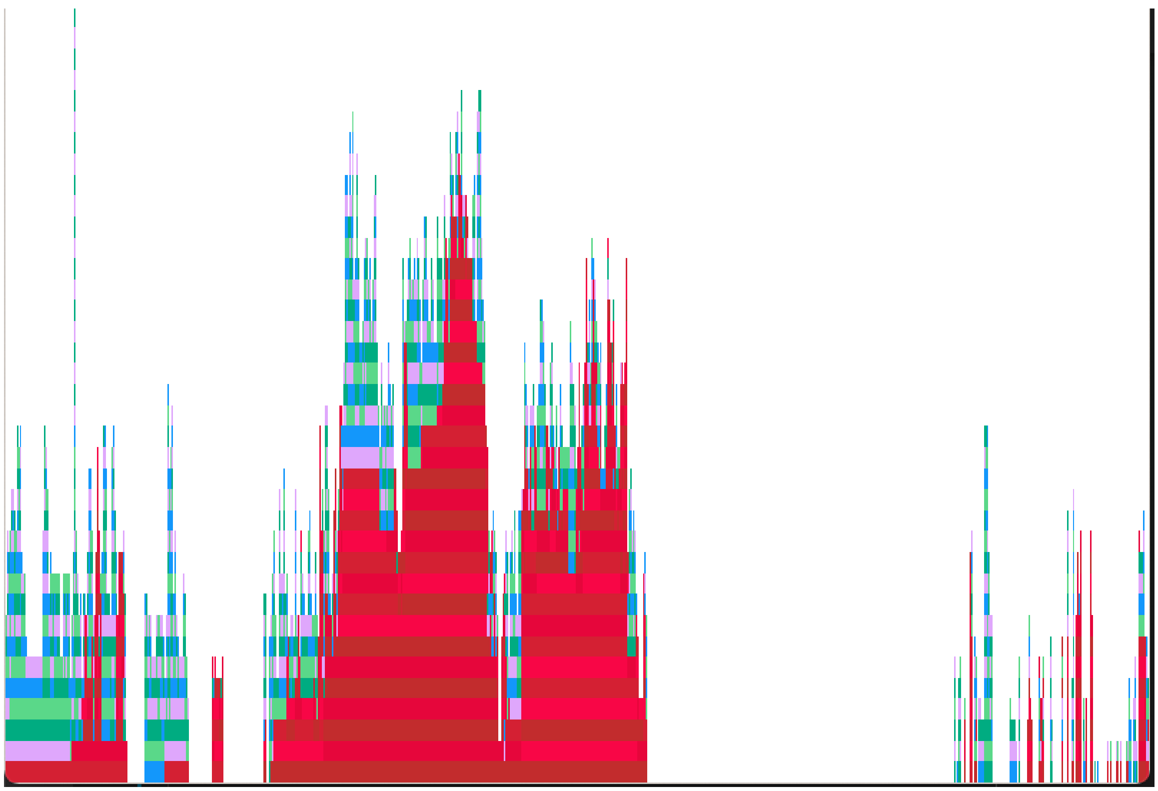

Figure 8.1: flamegraph

Part III

Manual

Chapter 9

Standard form problem

MathOptInterface represents optimization problems in the standard form:

$$\min_{x \in \mathbb{R}^n} \quad f_0(x) \tag{9.1}$$

$$\text{s.t.} \quad f_i(x) \in \mathcal{S}_i \qquad i = 1 \ldots m \tag{9.2}$$

where:

- the functions $f_0, f_1, \ldots, f_m$ are specified by `AbstractFunction` objects
- the sets $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are specified by `AbstractSet` objects

Tip

For more information on this standard form, read our paper.

MOI defines some commonly used functions and sets, but the interface is extensible to other sets recognized by the solver.

9.1 Functions

The function types implemented in MathOptInterface.jl are:

- `VariableIndex`: x_j, i.e., projection onto a single coordinate defined by a variable index j.
- `VectorOfVariables`: projection onto multiple coordinates (i.e., extracting a subvector).
- `ScalarAffineFunction`: $a^T x + b$, where a is a vector and b scalar.
- `VectorAffineFunction`: $Ax + b$, where A is a matrix and b is a vector.
- `ScalarQuadraticFunction`: $\frac{1}{2} x^T Q x + a^T x + b$, where Q is a symmetric matrix, a is a vector, and b is a constant.
- `VectorQuadraticFunction`: a vector of scalar-valued quadratic functions.

Extensions for nonlinear programming are present but not yet well documented.

9.2 One-dimensional sets

The one-dimensional set types implemented in MathOptInterface.jl are:

- `LessThan(upper)`: $\{x \in \mathbb{R} : x \le \text{upper}\}$
- `GreaterThan(lower)`: $\{x \in \mathbb{R} : x \ge \text{lower}\}$
- `EqualTo(value)`: $\{x \in \mathbb{R} : x = \text{value}\}$
- `Interval(lower, upper)`: $\{x \in \mathbb{R} : x \in [\text{lower}, \text{upper}]\}$
- `Integer()`: $\mathbb{Z}$
- `ZeroOne()`: $\{0, 1\}$
- `Semicontinuous(lower,upper)`: $\{0\} \cup [\text{lower}, \text{upper}]$
- `Semiinteger(lower,upper)`: $\{0\} \cup \{\text{lower}, \text{lower} + 1, \ldots, \text{upper} - 1, \text{upper}\}$

9.3 Vector cones

The vector-valued set types implemented in MathOptInterface.jl are:

- `Reals(dimension)`: $\mathbb{R}^{\text{dimension}}$
- `Zeros(dimension)`: $0^{\text{dimension}}$
- `Nonnegatives(dimension)`: $\{x \in \mathbb{R}^{\text{dimension}} : x \ge 0\}$
- `Nonpositives(dimension)`: $\{x \in \mathbb{R}^{\text{dimension}} : x \le 0\}$
- `SecondOrderCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \ge \|x\|_2\}$
- `RotatedSecondOrderCone(dimension)`: $\{(t, u, x) \in \mathbb{R}^{\text{dimension}} : 2tu \ge \|x\|_2^2, t, u \ge 0\}$
- `ExponentialCone()`: $\{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \le z, y > 0\}$
- `DualExponentialCone()`: $\{(u, v, w) \in \mathbb{R}^3 : -u \exp(v/u) \le \exp(1) w, u < 0\}$
- `GeometricMeanCone(dimension)`: $\{(t, x) \in \mathbb{R}^{n+1} : x \ge 0, t \le \sqrt[n]{x_1 x_2 \cdots x_n}\}$ where n is dimension-1
- `PowerCone(exponent)`: $\{(x, y, z) \in \mathbb{R}^3 : x^{\text{exponent}} y^{1-\text{exponent}} \ge |z|, x, y \ge 0\}$
- `DualPowerCone(exponent)`: $\{(u, v, w) \in \mathbb{R}^3 : \frac{u}{\text{exponent}}^{\text{exponent}} \frac{v}{1-\text{exponent}}^{1-\text{exponent}} \ge |w|, u, v \ge 0\}$
- `NormOneCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \ge \|x\|_1\}$ where $\|x\|_1 = \sum_i |x_i|$
- `NormInfinityCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \ge \|x\|_\infty\}$ where $\|x\|_\infty = \max_i |x_i|$.
- `RelativeEntropyCone(dimension)`: $\{(u, v, w) \in \mathbb{R}^{\text{dimension}} : u \ge \sum_i w_i \log(\frac{w_i}{v_i}), v_i \ge 0, w_i \ge 0\}$

9.4 Matrix cones

The matrix-valued set types implemented in MathOptInterface.jl are:

- `RootDetConeTriangle(dimension)`: $\{(t, X) \in \mathbb{R}^{1+\text{dimension}(1+\text{dimension})/2} : t \le \det(X)^{1/\text{dimension}}, X$ is the upper
- `RootDetConeSquare(dimension)`: $\{(t, X) \in \mathbb{R}^{1+\text{dimension}^2} : t \le \det(X)^{1/\text{dimension}}, X$ is a PSD matrix$\}$
- `PositiveSemidefiniteConeTriangle(dimension)`: $\{X \in \mathbb{R}^{\text{dimension}(\text{dimension}+1)/2} : X$ is the upper triangle of a PSI
- `PositiveSemidefiniteConeSquare(dimension)`: $\{X \in \mathbb{R}^{\text{dimension}^2} : X$ is a PSD matrix$\}$
- `LogDetConeTriangle(dimension)`: $\{(t, u, X) \in \mathbb{R}^{2+\text{dimension}(1+\text{dimension})/2} : t \le u \log(\det(X/u)), X$ is the upper t
$0\}$
- `LogDetConeSquare(dimension)`: $\{(t, u, X) \in \mathbb{R}^{2+\text{dimension}^2} : t \le u \log(\det(X/u)), X$ is a PSD matrix, $u >$
$0\}$
- `NormSpectralCone(row_dim, column_dim)`: $\{(t, X) \in \mathbb{R}^{1+\text{row\_dim}\times\text{column\_dim}} : t \ge \sigma_1(X), X$ is a matrix with row_c
- `NormNuclearCone(row_dim, column_dim)`: $\{(t, X) \in \mathbb{R}^{1+\text{row\_dim}\times\text{column\_dim}} : t \ge \sum_i \sigma_i(X), X$ is a matrix with row

Some of these cones can take two forms: `XXXConeTriangle` and `XXXConeSquare`.

In `XXXConeTriangle` sets, the matrix is assumed to be symmetric, and the elements are provided by a vector, in which the entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row).

In `XXXConeSquare` sets, the entries of the matrix are given column by column (or equivalently, row by row), and the matrix is constrained to be symmetric. As an example, given a 2-by-2 matrix of variables X and a one-dimensional variable `t`, we can specify a root-det constraint as `[t, X11, X12, X22] ∈ RootDetConeTriangle` or `[t, X11, X12, X21, X22] ∈ RootDetConeSquare`.

We provide both forms to enable flexibility for solvers who may natively support one or the other. Transformations between `XXXConeTriangle` and `XXXConeSquare` are handled by bridges, which removes the chance of conversion mistakes by users or solver developers.

9.5 Multi-dimensional sets with combinatorial structure

- `SOS1(weights)`: A special ordered set of Type I.
- `SOS2(weights)`: A special ordered set of Type II.
- `Indicator(set)`: A set to specify indicator constraints.
- `Complements(dimension)`: A set for mixed complementarity constraints.

Chapter 10

Models

The most significant part of MOI is the definition of the **model API** that is used to specify an instance of an optimization problem (e.g., by adding variables and constraints). Objects that implement the model API must inherit from the `ModelLike` abstract type.

Notably missing from the model API is the method to solve an optimization problem. `ModelLike` objects may store an instance (e.g., in memory or backed by a file format) without being linked to a particular solver. In addition to the model API, MOI defines `AbstractOptimizer` and provides methods to solve the model and interact with solutions. See the Solutions section for more details.

Info

Throughout the rest of the manual, `model` is used as a generic `ModelLike`, and `optimizer` is used as a generic `AbstractOptimizer`.

Tip

MOI does not export functions, but for brevity we often omit qualifying names with the MOI module. Best practice is to have

```julia
using MathOptInterface
const MOI = MathOptInterface
```

and prefix all MOI methods with `MOI.` in user code. If a name is also available in base Julia, we always explicitly use the module prefix, for example, with `MOI.get`.

10.1 Attributes

Attributes are properties of the model that can be queried and modified. These include constants such as the number of variables in a model `NumberOfVariables`), and properties of variables and constraints such as the name of a variable (`VariableName`).

There are four types of attributes:

- Model attributes (subtypes of `AbstractModelAttribute`) refer to properties of a model.
- Optimizer attributes (subtypes of `AbstractOptimizerAttribute`) refer to properties of an optimizer.
- Constraint attributes (subtypes of `AbstractConstraintAttribute`) refer to properties of an individual constraint.

- Variable attributes (subtypes of `AbstractVariableAttribute`) refer to properties of an individual variable.

Some attributes are values that can be queried by the user but not modified, while other attributes can be modified by the user.

All interactions with attributes occur through the `get` and `set` functions.

Consult the docstsrings of each attribute for information on what it represents.

10.2 ModelLike API

The following attributes are available:

- `ListOfConstraintAttributesSet`
- `ListOfConstraintIndices`
- `ListOfConstraintTypesPresent`
- `ListOfModelAttributesSet`
- `ListOfVariableAttributesSet`
- `ListOfVariableIndices`
- `NumberOfConstraints`
- `NumberOfVariables`
- `Name`
- `ObjectiveFunction`
- `ObjectiveFunctionType`
- `ObjectiveSense`

10.3 AbstractOptimizer API

The following attributes are available:

- `DualStatus`
- `PrimalStatus`
- `RawStatusString`
- `ResultCount`
- `TerminationStatus`
- `BarrierIterations`
- `DualObjectiveValue`
- `NodeCount`

- `NumberOfThreads`
- `ObjectiveBound`
- `ObjectiveValue`
- `RelativeGap`
- `RawOptimizerAttribute`
- `RawSolver`
- `Silent`
- `SimplexIterations`
- `SolverName`
- `SolverVersion`
- `SolveTimeSec`
- `TimeLimitSec`

Chapter 11

Variables

11.1 Add a variable

Use add_variable to add a single variable.

```
julia> x = MOI.add_variable(model)
MathOptInterface.VariableIndex(1)
```

add_variable returns a VariableIndex type, which is used to refer to the added variable in other calls.

Check if a VariableIndex is valid using is_valid.

```
julia> MOI.is_valid(model, x)
true
```

Use add_variables to add a number of variables.

```
julia> y = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
 MathOptInterface.VariableIndex(2)
 MathOptInterface.VariableIndex(3)
```

Warning

The integer does not necessarily corresond to the column inside an optimizer!

11.2 Delete a variable

Delete a variable using delete.

```
julia> MOI.delete(model, x)

julia> MOI.is_valid(model, x)
false
```

Warning

Not all `ModelLike` models support deleting variables. A DeleteNotAllowed error is thrown if this is not supported.

11.3 Variable attributes

The following attributes are available for variables:

- VariableName
- VariablePrimalStart
- VariablePrimal

Get and set these attributes using get and set.

```
julia> MOI.set(model, MOI.VariableName(), x, "var_x")

julia> MOI.get(model, MOI.VariableName(), x)
"var_x"
```

Chapter 12

Constraints

12.1 Add a constraint

Use `add_constraint` to add a single constraint.

```
julia> c = MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Nonnegatives(2))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.Nonnegatives}(1)
```

`add_constraint` returns a `ConstraintIndex` type, which is used to refer to the added constraint in other calls.

Check if a `ConstraintIndex` is valid using `is_valid`.

```
julia> MOI.is_valid(model, c)
true
```

Use `add_constraints` to add a number of constraints of the same type.

```
julia> c = MOI.add_constraints(
           model,
           [x[1], x[2]],
           [MOI.GreaterThan(0.0), MOI.GreaterThan(1.0)]
       )
2-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}(1)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}(2)
```

This time, a vector of `ConstraintIndex` are returned.

Use `supports_constraint` to check if the model supports adding a constraint type.

```
julia> MOI.supports_constraint(
           model,
           MOI.VariableIndex,
           MOI.GreaterThan{Float64},
       )
true
```

12.2 Delete a constraint

Use delete to delete a constraint.

```
julia> MOI.delete(model, c)

julia> MOI.is_valid(model, c)
false
```

12.3 Constraint attributes

The following attributes are available for constraints:

- ConstraintName
- ConstraintPrimalStart
- ConstraintDualStart
- ConstraintPrimal
- ConstraintDual
- ConstraintBasisStatus
- ConstraintFunction
- CanonicalConstraintFunction
- ConstraintSet

Get and set these attributes using get and set.

```
julia> MOI.set(model, MOI.ConstraintName(), c, "con_c")

julia> MOI.get(model, MOI.ConstraintName(), c)
"con_c"
```

12.4 Constraints by function-set pairs

Below is a list of common constraint types and how they are represented as function-set pairs in MOI. In the notation below, x is a vector of decision variables, x_i is a scalar decision variable, α, β are scalar constants, a, b are constant vectors, A is a constant matrix and $\mathbb{R}_+$ (resp. $\mathbb{R}_-$) is the set of nonnegative (resp. nonpositive) real numbers.

Linear constraints

By convention, solvers are not expected to support nonzero constant terms in the ScalarAffineFunctions the first four rows above, because they are redundant with the parameters of the sets. For example, encode $2x + 1 \leq 2$ as $2x \leq 1$.

Constraints with VariableIndex in LessThan, GreaterThan, EqualTo, or Interval sets have a natural interpretation as variable bounds. As such, it is typically not natural to impose multiple lower- or upper-bounds on the same variable, and the solver interfaces will throw respectively LowerBoundAlreadySet or UpperBoundAlreadySet.

Mathematical Constraint	MOI Function	MOI Set
$a^T x \le \beta$	`ScalarAffineFunction`	`LessThan`
$a^T x \ge \alpha$	`ScalarAffineFunction`	`GreaterThan`
$a^T x = \beta$	`ScalarAffineFunction`	`EqualTo`
$\alpha \le a^T x \le \beta$	`ScalarAffineFunction`	`Interval`
$x_i \le \beta$	`VariableIndex`	`LessThan`
$x_i \ge \alpha$	`VariableIndex`	`GreaterThan`
$x_i = \beta$	`VariableIndex`	`EqualTo`
$\alpha \le x_i \le \beta$	`VariableIndex`	`Interval`
$Ax + b \in \mathbb{R}_+^n$	`VectorAffineFunction`	`Nonnegatives`
$Ax + b \in \mathbb{R}_-^n$	`VectorAffineFunction`	`Nonpositives`
$Ax + b = 0$	`VectorAffineFunction`	`Zeros`

Moreover, adding two `VariableIndex` constraints on the same variable with the same set is impossible because they share the same index as it is the index of the variable, see `ConstraintIndex`.

It is natural, however, to impose upper- and lower-bounds separately as two different constraints on a single variable. The difference between imposing bounds by using a single `Interval` constraint and by using separate `LessThan` and `GreaterThan` constraints is that the latter will allow the solver to return separate dual multipliers for the two bounds, while the former will allow the solver to return only a single dual for the interval constraint.

Conic constraints

Mathematical Constraint	MOI Function	MOI Set
$\|Ax + b\|_2 \le c^T x + d$	`VectorAffineFunction`	`SecondOrderCone`
$y \ge \|x\|_2$	`VectorOfVariables`	`SecondOrderCone`
$2yz \ge \|x\|_2^2, y, z \ge 0$	`VectorOfVariables`	`RotatedSecondOrderCone`
$(a_1^T x + b_1, a_2^T x + b_2, a_3^T x + b_3) \in \mathcal{E}$	`VectorAffineFunction`	`ExponentialCone`
$A(x) \in \mathcal{S}_+$	`VectorAffineFunction`	`PositiveSemidefiniteConeTriangle`
$B(x) \in \mathcal{S}_+$	`VectorAffineFunction`	`PositiveSemidefiniteConeSquare`
$x \in \mathcal{S}_+$	`VectorOfVariables`	`PositiveSemidefiniteConeTriangle`
$x \in \mathcal{S}_+$	`VectorOfVariables`	`PositiveSemidefiniteConeSquare`

where $\mathcal{E}$ is the exponential cone (see `ExponentialCone`), $\mathcal{S}_+$ is the set of positive semidefinite symmetric matrices, A is an affine map that outputs symmetric matrices and B is an affine map that outputs square matrices.

Quadratic constraints

Mathematical Constraint	MOI Function	MOI Set
$x^T Q x + a^T x + b \ge 0$	`ScalarQuadraticFunction`	`GreaterThan`
$x^T Q x + a^T x + b \le 0$	`ScalarQuadraticFunction`	`LessThan`
$x^T Q x + a^T x + b = 0$	`ScalarQuadraticFunction`	`EqualTo`
Bilinear matrix inequality	`VectorQuadraticFunction`	`PositiveSemidefiniteCone...`

Discrete and logical constraints

12.5 JuMP mapping

The following bullet points show examples of how JuMP constraints are translated into MOI function-set pairs:

Mathematical Constraint	MOI Function	MOI Set
$x_i \in \mathbb{Z}$	`VariableIndex`	`Integer`
$x_i \in \{0, 1\}$	`VariableIndex`	`ZeroOne`
$x_i \in \{0\} \cup [l, u]$	`VariableIndex`	`Semicontinuous`
$x_i \in \{0\} \cup \{l, l + 1, \ldots, u - 1, u\}$	`VariableIndex`	`Semiinteger`
At most one component of x can be nonzero	`VectorOfVariables`	`SOS1`
At most two components of x can be nonzero, and if so they must be adjacent components	`VectorOfVariables`	`SOS2`
$y = 1 \implies a^T x \in S$	`VectorAffineFunction`	`Indicator`

- `@constraint(m, 2x + y <= 10)` becomes `ScalarAffineFunction`-in-`LessThan`
- `@constraint(m, 2x + y >= 10)` becomes `ScalarAffineFunction`-in-`GreaterThan`
- `@constraint(m, 2x + y == 10)` becomes `ScalarAffineFunction`-in-`EqualTo`
- `@constraint(m, 0 <= 2x + y <= 10)` becomes `ScalarAffineFunction`-in-`Interval`
- `@constraint(m, 2x + y in ArbitrarySet())` becomes `ScalarAffineFunction`-in-`ArbitrarySet`.

Variable bounds are handled in a similar fashion:

- `@variable(m, x <= 1)` becomes `VariableIndex`-in-`LessThan`
- `@variable(m, x >= 1)` becomes `VariableIndex`-in-`GreaterThan`

One notable difference is that a variable with an upper and lower bound is translated into two constraints, rather than an interval. i.e.:

- `@variable(m, 0 <= x <= 1)` becomes `VariableIndex`-in-`LessThan` and `VariableIndex`-in-`GreaterThan`.

Chapter 13

Solutions

13.1 Solving and retrieving the results

Once an optimizer is loaded with the objective function and all of the constraints, we can ask the solver to solve the model by calling `optimize!`.

```
MOI.optimize!(optimizer)
```

13.2 Why did the solver stop?

The optimization procedure may terminate for a number of reasons. The `TerminationStatus` attribute of the optimizer returns a `TerminationStatusCode` object which explains why the solver stopped.

The termination statuses distinguish between proofs of optimality, infeasibility, local convergence, limits, and termination because of something unexpected like invalid problem data or failure to converge.

A typical usage of the `TerminationStatus` attribute is as follows:

```
status = MOI.get(optimizer, TerminationStatus())
if status == MOI.OPTIMAL
    # Ok, we solved the problem!
else
    # Handle other cases.
end
```

After checking the `TerminationStatus`, check `ResultCount`. This attribute returns the number of results that the solver has available to return. A result is defined as a primal-dual pair, but either the primal or the dual may be missing from the result. While the `OPTIMAL` termination status normally implies that at least one result is available, other statuses do not. For example, in the case of infeasiblity, a solver may return no result or a proof of infeasibility. The `ResultCount` attribute distinguishes between these two cases.

13.3 Primal solutions

Use the `PrimalStatus` optimizer attribute to return a `ResultStatusCode` describing the status of the primal solution.

Common returns are described below in the Common status situations section.

Query the primal solution using the `VariablePrimal` and `ConstraintPrimal` attributes.

Query the objective function value using the `ObjectiveValue` attribute.

13.4 Dual solutions

Warning

See Duality for a discussion of the MOI conventions for primal-dual pairs and certificates.

Use the `DualStatus` optimizer attribute to return a `ResultStatusCode` describing the status of the dual solution.

Query the dual solution using the `ConstraintDual` attribute.

Query the dual objective function value using the `DualObjectiveValue` attribute.

13.5 Common status situations

The sections below describe how to interpret typical or interesting status cases for three common classes of solvers. The example cases are illustrative, not comprehensive. Solver wrappers may provide additional information on how the solver's statuses map to MOI statuses.

Info

* in the tables indicate that multiple different values are possible.

Primal-dual convex solver

Linear programming and conic optimization solvers fall into this category.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Proved optimality	OPTIMAL	1	FEASIBLE_POINT	FEASIBLE_POINT
Proved infeasible	INFEASIBLE	1	NO_SOLUTION	INFEASIBILITY_CERTIFICATE
Optimal within relaxed tolerances	ALMOST_OPTIMAL	1	FEASIBLE_POINT	FEASIBLE_POINT
Optimal within relaxed tolerances	ALMOST_OPTIMAL	1	ALMOST_FEASIBLE_POINT	ALMOST_FEASIBLE_POINT
Detected an unbounded ray of the primal	DUAL_INFEASIBLE	1	INFEASIBILITY_CERTIFICATE	NO_SOLUTION
Stall	SLOW_PROGRESS	1	*	*

Global branch-and-bound solvers

Mixed-integer programming solvers fall into this category.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Proved optimality	OPTIMAL	1	FEASIBLE_POINT	NO_SOLUTION
Presolve detected infeasibility or unboundedness	INFEASIBLE_OR_UNBOUNDED	0	NO_SOLUTION	NO_SOLUTION
Proved infeasibility	INFEASIBLE	0	NO_SOLUTION	NO_SOLUTION
Timed out (no solution)	TIME_LIMIT	0	NO_SOLUTION	NO_SOLUTION
Timed out (with a solution)	TIME_LIMIT	1	FEASIBLE_POINT	NO_SOLUTION
CPXMIP_OPTIMAL_INFEAS	ALMOST_OPTIMAL	1	INFEASIBLE_POINT	NO_SOLUTION

Info

CPXMIP_OPTIMAL_INFEAS is a CPLEX status that indicates that a preprocessed problem was solved to optimality, but the solver was unable to recover a feasible solution to the original problem. Handling this status was one of the motivating drivers behind the design of MOI.

Local search solvers

Nonlinear programming solvers fall into this category. It also includes non-global tree search solvers like Juniper.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Converged to a stationary point	LOCALLY_SOLVED	1	FEASIBLE_POINT	FEASIBLE_POINT
Completed a non-global tree search (with a solution)	LOCALLY_SOLVED	1	FEASIBLE_POINT	FEASIBLE_POINT
Converged to an infeasible point	LOCALLY_INFEASIBLE	1	INFEASIBLE_POINT	*
Completed a non-global tree search (no solution found)	LOCALLY_INFEASIBLE	0	NO_SOLUTION	NO_SOLUTION
Iteration limit	ITERATION_LIMIT	1	*	*
Diverging iterates	NORM_LIMIT or OBJECTIVE_LIMIT	1	*	*

Chapter 14

Problem modification

In addition to adding and deleting constraints and variables, MathOptInterface supports modifying, in-place, coefficients in the constraints and the objective function of a model.

These modifications can be grouped into two categories:

- modifications which replace the set of function of a constraint with a new set or function
- modifications which change, in-place, a component of a function

Warning

Solve `ModelLike` objects do not support problem modification.

14.1 Modify the set of a constraint

Use `set` and `ConstraintSet` to modify the set of a constraint by replacing it with a new instance of the same type.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.EqualTo(2.0));

julia> MOI.get(model, MOI.ConstraintSet(), c) == MOI.EqualTo(2.0)
true
```

However, the following will fail as the new set is of a different type to the original set:

```
julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.GreaterThan(2.0))
ERROR: [...]
```

Special cases: set transforms

If our constraint is an affine inequality, then this corresponds to modifying the right-hand side of a constraint in linear programming.

In some special cases, solvers may support efficiently changing the set of a constraint (for example, from `LessThan` to `GreaterThan`). For these cases, MathOptInterface provides the `transform` method.

The `transform` function returns a new constraint index, and the old constraint index (i.e., c) is no longer valid.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.LessThan(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}(1)

julia> new_c = MOI.transform(model, c, MOI.GreaterThan(2.0));

julia> MOI.is_valid(model, c)
false

julia> MOI.is_valid(model, new_c)
true
```

> **Note**
>
> `transform` cannot be called with a set of the same type. Use `set` instead.

14.2 Modify the function of a constraint

Use `set` and `ConstraintFunction` to modify the function of a constraint by replacing it with a new instance of the same type.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(2.0, x)], 1.0);

julia> MOI.set(model, MOI.ConstraintFunction(), c, new_f);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

However, the following will fail as the new function is of a different type to the original function:

```
julia> MOI.set(model, MOI.ConstraintFunction(), c, x)
ERROR: [...]
```

14.3 Modify constant term in a scalar function

Use modify and ScalarConstantChange to modify the constant term in a ScalarAffineFunction or ScalarQuadraticFunction.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> MOI.modify(model, c, MOI.ScalarConstantChange(1.0));

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 1.0);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

Tip

ScalarConstantChange can also be used to modify the objective function by passing an instance of ObjectiveFunction instead of the constraint index c as we saw above.

```
julia> MOI.set(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           new_f,
       );

julia> MOI.modify(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           MOI.ScalarConstantChange(-1.0)
       );

julia> MOI.get(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
       ) ≈ MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], -1.0)
true
```

14.4 Modify constant terms in a vector function

Use modify and VectorConstantChange to modify the constant vector in a VectorAffineFunction or VectorQuadraticFunction

```
julia> c = MOI.add_constraint(
           model,
           MOI.VectorAffineFunction([
                   MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
                   MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
               ],
               [0.0, 0.0],
           ),
```

```
           MOI.Nonnegatives(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪  MathOptInterface.Nonnegatives}(1)

julia> MOI.modify(model, c, MOI.VectorConstantChange([3.0, 4.0]));

julia> new_f = MOI.VectorAffineFunction(
           [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
           ],
           [3.0, 4.0],
       );

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

14.5 Modify affine coefficients in a scalar function

Use modify and ScalarCoefficientChange to modify the affine coefficient of a ScalarAffineFunction or ScalarQuadraticFunction.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> MOI.modify(model, c, MOI.ScalarCoefficientChange(x, 2.0));

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(2.0, x)], 0.0);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

Tip

ScalarCoefficientChange can also be used to modify the objective function by passing an instance of ObjectiveFunction instead of the constraint index c as we saw above.

14.6 Modify affine coefficients in a vector function

Use modify and MultirowChange to modify a vector of affine coefficients in a VectorAffineFunction or a VectorQuadraticFunction.

```
julia> c = MOI.add_constraint(
           model,
           MOI.VectorAffineFunction([
                   MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
                   MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
               ],
```

```
                [0.0, 0.0],
            ),
            MOI.Nonnegatives(2),
        )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪  MathOptInterface.Nonnegatives}(1)

julia> MOI.modify(model, c, MOI.MultirowChange(x, [(1, 3.0), (2, 4.0)]));

julia> new_f = MOI.VectorAffineFunction(
            [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(3.0, x)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(4.0, x)),
            ],
            [0.0, 0.0],
        );

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

Part IV

Background

Chapter 15

Duality

Conic duality is the starting point for MOI's duality conventions. When all functions are affine (or coordinate projections), and all constraint sets are closed convex cones, the model may be called a conic optimization problem.

For a minimization problem in geometric conic form, the primal is:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{15.1}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m \tag{15.2}$$

and the dual is a maximization problem in standard conic form:

$$\max_{y_1, \ldots, y_m} \quad -\sum_{i=1}^m b_i^T y_i + b_0 \tag{15.3}$$

$$\text{s.t.} \quad a_0 - \sum_{i=1}^m A_i^T y_i = 0 \tag{15.4}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m \tag{15.5}$$

where each $\mathcal{C}_i$ is a closed convex cone and $\mathcal{C}_i^*$ is its dual cone.

For a maximization problem in geometric conic form, the primal is:

$$\max_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{15.6}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m \tag{15.7}$$

and the dual is a minimization problem in standard conic form:

$$\min_{y_1, \ldots, y_m} \quad \sum_{i=1}^m b_i^T y_i + b_0 \tag{15.8}$$

$$\text{s.t.} \quad a_0 + \sum_{i=1}^m A_i^T y_i = 0 \tag{15.9}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m \tag{15.10}$$

A linear inequality constraint $a^T x + b \geq c$ is equivalent to $a^T x + b - c \in \mathbb{R}_+$, and $a^T x + b \leq c$ is equivalent to $a^T x + b - c \in \mathbb{R}_-$. Variable-wise constraints are affine constraints with the appropriate identity mapping in place of A_i.

For the special case of minimization LPs, the MOI primal form can be stated as:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{15.11}$$
$$\text{s.t.} \quad A_1 x \geq b_1 \tag{15.12}$$
$$A_2 x \leq b_2 \tag{15.13}$$
$$A_3 x = b_3 \tag{15.14}$$

By applying the stated transformations to conic form, taking the dual, and transforming back into linear inequality form, one obtains the following dual:

$$\max_{y_1, y_2, y_3} \quad b_1^T y_1 + b_2^T y_2 + b_3^T y_3 + b_0 \tag{15.15}$$
$$\text{s.t.} \quad A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = a_0 \tag{15.16}$$
$$y_1 \geq 0 \tag{15.17}$$
$$y_2 \leq 0 \tag{15.18}$$

For maximization LPs, the MOI primal form can be stated as:

$$\max_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{15.19}$$
$$\text{s.t.} \quad A_1 x \geq b_1 \tag{15.20}$$
$$A_2 x \leq b_2 \tag{15.21}$$
$$A_3 x = b_3 \tag{15.22}$$

and similarly, the dual is:

$$\min_{y_1, y_2, y_3} \quad -b_1^T y_1 - b_2^T y_2 - b_3^T y_3 + b_0 \tag{15.23}$$
$$\text{s.t.} \quad A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = -a_0 \tag{15.24}$$
$$y_1 \geq 0 \tag{15.25}$$
$$y_2 \leq 0 \tag{15.26}$$

> **Warning**
>
> For the LP case, the signs of the feasible dual variables depend only on the sense of the corresponding primal inequality and not on the objective sense.

15.1 Duality and scalar product

The scalar product is different from the canonical one for the sets `PositiveSemidefiniteConeTriangle`, `LogDetConeTriangle`, `RootDetConeTriangle`.

If the set C_i of the section Duality is one of these three cones, then the rows of the matrix A_i corresponding to off-diagonal entries are twice the value of the `coefficients` field in the `VectorAffineFunction` for the corresponding rows. See `PositiveSemidefiniteConeTriangle` for details.

15.2 Dual for problems with quadratic functions

Quadratic Programs (QPs)

For quadratic programs with only affine conic constraints,

$$
\begin{array}{lll}
\min_{x\in\mathbb{R}^n} & \frac{1}{2}x^TQ_0x + a_0^Tx + b_0 & \\
\text{s.t.} & A_ix + b_i \in \mathcal{C}_i & i = 1\dots m.
\end{array}
$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}^{m_i}$ for $i = 1,\dots,m$, consider the Lagrangian function

$$L(x,y) = \frac{1}{2}x^TQ_0x + a_0^Tx + b_0 - \sum_{i=1}^m y_i^T(A_ix + b_i).$$

Let $z(y)$ denote $\sum_{i=1}^m A_i^Ty_i - a_0$, the Lagrangian can be rewritten as

$$L(x,y) = \frac{1}{2}x^TQ_0x - z(y)^Tx + b_0 - \sum_{i=1}^m y_i^Tb_i.$$

The condition $\nabla_x L(x,y) = 0$ gives

$$0 = \nabla_x L(x,y) = Q_0x + a_0 - \sum_{i=1}^m y_i^Tb_i$$

which gives $Q_0x = z(y)$. This allows to obtain that

$$\min_{x\in\mathbb{R}^n} L(x,y) = -\frac{1}{2}x^TQ_0x + b_0 - \sum_{i=1}^m y_i^Tb_i$$

so the dual problem is

$$\max_{y_i\in\mathcal{C}_i^*}\min_{x\in\mathbb{R}^n} -\frac{1}{2}x^TQ_0x + b_0 - \sum_{i=1}^m y_i^Tb_i.$$

If Q_0 is invertible, we have $x = Q_0^{-1}z(y)$ hence

$$\min_{x\in\mathbb{R}^n} L(x,y) = -\frac{1}{2}z(y)^TQ_0^{-1}z(y) + b_0 - \sum_{i=1}^m y_i^Tb_i$$

so the dual problem is

$$\max_{y_i\in\mathcal{C}_i^*} -\frac{1}{2}z(y)^TQ_0^{-1}z(y) + b_0 - \sum_{i=1}^m y_i^Tb_i.$$

Quadratically Constrained Quadratic Programs (QCQPs)

Given a problem with both quadratic function and quadratic objectives:

$$\begin{array}{lll} \min\limits_{x\in\mathbb{R}^n} & \frac{1}{2}x^TQ_0x + a_0^Tx + b_0 & \\ \text{s.t.} & \frac{1}{2}x^TQ_ix + a_i^Tx + b_i \in \mathcal{C}_i & i = 1\ldots m. \end{array}$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}$ for $i = 1\ldots m$, consider the Lagrangian function

$$L(x,y) = \frac{1}{2}x^TQ_0x + a_0^Tx + b_0 - \sum_{i=1}^m y_i(\frac{1}{2}x^TQ_ix + a_i^Tx + b_i)$$

A pair of primal-dual variables $(x^\star, y^\star)$ is optimal if

- $x^\star$ is a minimizer of

  $$\min_{x\in\mathbb{R}^n} L(x,y^\star).$$

  That is,

  $$0 = \nabla_x L(x,y^\star) = Q_0x + a_0 - \sum_{i=1}^m y_i^\star(Q_ix + a_i).$$

- and $y^\star$ is a maximizer of

  $$\max_{y_i\in\mathcal{C}_i^*} L(x^\star, y).$$

  That is, for all $i = 1,\ldots,m$, $\frac{1}{2}x^TQ_ix + a_i^Tx + b_i$ is either zero or in the normal cone of $\mathcal{C}_i^*$ at $y^\star$. For instance, if $\mathcal{C}_i$ is $\{z\in\mathbb{R} : z \leq 0\}$, this means that if $\frac{1}{2}x^TQ_ix + a_i^Tx + b_i$ is nonzero at $x^\star$ then $y_i^\star = 0$. This is the classical complementary slackness condition.

If $\mathcal{C}_i$ is a vector set, the discussion remains valid with $y_i(\frac{1}{2}x^TQ_ix + a_i^Tx + b_i)$ replaced with the scalar product between y_i and the vector of scalar-valued quadratic functions.

Chapter 16

Infeasibility certificates

When given a conic problem that is infeasible or unbounded, some solvers can produce a certificate of infeasibility. This page explains what a certificate of infeasibility is, and the related conventions that MathOptInterface adopts.

16.1 Conic duality

MathOptInterface uses conic duality to define infeasibility certificates. A full explanation is given in the section Duality, but here is a brief overview.

Minimization problems

For a minimization problem in geometric conic form, the primal is:

$$
\begin{aligned}
&\min_{x \in \mathbb{R}^n} & & a_0^\top x + b_0 & & & (16.1)\\
&\text{s.t.} & & A_i x + b_i \in \mathcal{C}_i & & i = 1 \ldots m, & (16.2)
\end{aligned}
$$

and the dual is a maximization problem in standard conic form:

$$
\begin{aligned}
&\max_{y_1,\ldots,y_m} & & -\sum_{i=1}^m b_i^\top y_i + b_0 & & & (16.3)\\
&\text{s.t.} & & a_0 - \sum_{i=1}^m A_i^\top y_i = 0 & & & (16.4)\\
& & & y_i \in \mathcal{C}_i^* & & i = 1 \ldots m, & (16.5)
\end{aligned}
$$

where each $\mathcal{C}_i$ is a closed convex cone and $\mathcal{C}_i^*$ is its dual cone.

Maximization problems

For a maximization problem in geometric conic form, the primal is:

$$
\begin{aligned}
&\max_{x \in \mathbb{R}^n} & & a_0^\top x + b_0 & & & (16.6)\\
&\text{s.t.} & & A_i x + b_i \in \mathcal{C}_i & & i = 1 \ldots m, & (16.7)
\end{aligned}
$$

and the dual is a minimization problem in standard conic form:

$$\min_{y_1,\ldots,y_m} \quad \sum_{i=1}^{m} b_i^\top y_i + b_0 \tag{16.8}$$

$$\text{s.t.} \quad a_0 + \sum_{i=1}^{m} A_i^\top y_i = 0 \tag{16.9}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m. \tag{16.10}$$

16.2 Unbounded problems

A problem is unbounded if and only if:

1. there exists a feasible primal solution
2. the dual is infeasible.

A feasible primal solution—if one exists—can be obtained by setting `ObjectiveSense` to `FEASIBILITY_SENSE` before optimizing. Therefore, most solvers terminate after they prove the dual is infeasible via a certificate of dual infeasibility, but before they have found a feasible primal solution. This is also the reason that MathOptInterface defines the `DUAL_INFEASIBLE` status instead of `UNBOUNDED`.

A certificate of dual infeasibility is an improving ray of the primal problem. That is, there exists some vector d such that for all $\eta > 0$:

$$A_i(x + \eta d) + b_i \in \mathcal{C}_i, \ \ i = 1 \ldots m,$$

and (for minimization problems):

$$a_0^\top (x + \eta d) + b_0 < a_0^\top x + b_0,$$

for any feasible point x. The latter simplifies to $a_0^\top d < 0$. For maximization problems, the inequality is reversed, so that $a_0^\top d > 0$.

If the solver has found a certificate of dual infeasibility:

- `TerminationStatus` must be `DUAL_INFEASIBLE`
- `PrimalStatus` must be `INFEASIBILITY_CERTIFICATE`
- `VariablePrimal` must be the corresponding value of d
- `ConstraintPrimal` must be the corresponding value of $A_i d$
- `ObjectiveValue` must be the value $a_0^\top d$. Note that this is the value of the objective function at d, ignoring the constant `b_0`.

  Note

  The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

16.3 Infeasible problems

A certificate of primal infeasibility is an improving ray of the dual problem. However, because infeasibility is independent of the objective function, we first homogenize the primal problem by removing its objective.

For a minimization problem, a dual improving ray is some vector d such that for all $\eta > 0$:

$$-\sum_{i=1}^{m} A_i^\top (y_i + \eta d_i) = 0 \tag{16.11}$$

$$(y_i + \eta d_i) \in \mathcal{C}_i^* \qquad i = 1 \ldots m, \tag{16.12}$$

and:

$$-\sum_{i=1}^{m} b_i^\top (y_i + \eta d_i) > -\sum_{i=1}^{m} b_i^\top y_i,$$

for any feasible dual solution y. The latter simplifies to $-\sum_{i=1}^{m} b_i^\top d_i > 0$. For a maximization problem, the inequality is $\sum_{i=1}^{m} b_i^\top d_i < 0$. (Note that these are the same inequality, modulo a `-` sign.)

If the solver has found a certificate of primal infeasibility:

- `TerminationStatus` must be `INFEASIBLE`
- `DualStatus` must be `INFEASIBILITY_CERTIFICATE`
- `ConstraintDual` must be the corresponding value of d
- `DualObjectiveValue` must be the value $-\sum_{i=1}^{m} b_i^\top d_i$ for minimization problems and $\sum_{i=1}^{m} b_i^\top d_i$ for maximization problems.

Note

The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

Infeasibility certificates of variable bounds

Many linear solvers (e.g., Gurobi) do not provide explicit access to the primal infeasibility certificate of a variable bound. However, given a set of linear constraints:

$$l_A \le Ax \le u_A \tag{16.13}$$

$$l_x \le x \le u_x, \tag{16.14}$$

the primal certificate of the variable bounds can be computed using the primal certificate associated with the affine constraints, d. (Note that d will have one element for each row of the A matrix, and that some or all of the elements in the vectors l_A and u_A may be $\pm\infty$. If both l_A and u_A are finite for some row, the corresponding element in ‘d must be `0`.)

Given d, compute $\bar{d} = d^\top A$. If the bound is finite, a certificate for the lower variable bound of x_i is $\max\{\bar{d}_i, 0\}$, and a certificate for the upper variable bound is $\min\{\bar{d}_i, 0\}$.

Chapter 17

Naming conventions

MOI follows several conventions for naming functions and structures. These should also be followed by packages extending MOI.

17.1 Sets

Sets encode the structure of constraints. Their names should follow the following conventions:

- Abstract types in the set hierarchy should begin with `Abstract` and end in `Set`, e.g., `AbstractScalarSet`, `AbstractVectorSet`.
- Vector-valued conic sets should end with `Cone`, e.g., `NormInfinityCone`, `SecondOrderCone`.
- Vector-valued Cartesian products should be plural and not end in `Cone`, e.g., `Nonnegatives`, not `NonnegativeCone`.
- Matrix-valued conic sets should provide two representations: `ConeSquare` and `ConeTriangle`, e.g., `RootDetConeTriangle` and `RootDetConeSquare`. See Matrix cones for more details.
- Scalar sets should be singular, not plural, e.g., `Integer`, not `Integers`.
- As much as possible, the names should follow established conventions in the domain where this set is used: for instance, convex sets should have names close to those of CVX, and constraint-programming sets should follow MiniZinc's constraints.

Part V

API Reference

Chapter 18

Standard form

18.1 Functions

MathOptInterface.AbstractFunction – Type.

```
AbstractFunction
```

Abstract supertype for function objects.

source

MathOptInterface.AbstractScalarFunction – Type.

```
AbstractScalarFunction
```

Abstract supertype for scalar-valued function objects.

source

MathOptInterface.AbstractVectorFunction – Type.

```
AbstractVectorFunction
```

Abstract supertype for vector-valued function objects.

source

MathOptInterface.VariableIndex – Type.

```
VariableIndex
```

A type-safe wrapper for `Int64` for use in referencing variables in a model. To allow for deletion, indices need not be consecutive.

source

MathOptInterface.VectorOfVariables – Type.

```
VectorOfVariables(variables)
```

The function that extracts the vector of variables referenced by `variables`, a `Vector{VariableIndex}`. This function is naturally be used for constraints that apply to groups of variables, such as an "all different" constraint, an indicator constraint, or a complementarity constraint.

source

MathOptInterface.ScalarAffineTerm – Type.

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

Represents cx_i where c is `coefficient` and x_i is the variable identified by `variable`.

source

MathOptInterface.ScalarAffineFunction – Type.

```
ScalarAffineFunction{T}(terms, constant)
```

The scalar-valued affine function $a^T x + b$, where:

- a is a sparse vector specified by a list of ScalarAffineTerm structs.
- b is a scalar specified by `constant::T`

Duplicate variable indices in `terms` are accepted, and the corresponding coefficients are summed together.

source

MathOptInterface.VectorAffineTerm – Type.

```
struct VectorAffineTerm{T}
    output_index::Int64
    scalar_term::ScalarAffineTerm{T}
end
```

A `ScalarAffineTerm` plus its index of the output component of a `VectorAffineFunction` or `VectorQuadraticFunction`. `output_index` can also be interpreted as a row index into a sparse matrix, where the `scalar_term` contains the column index and coefficient.

source

MathOptInterface.VectorAffineFunction – Type.

```
VectorAffineFunction{T}(terms, constants)
```

The vector-valued affine function $Ax + b$, where:

- A is a sparse matrix specified by a list of `VectorAffineTerm` objects.
- b is a vector specified by `constants`

Duplicate indices in the A are accepted, and the corresponding coefficients are summed together.

source

MathOptInterface.ScalarQuadraticTerm – Type.

```
struct ScalarQuadraticTerm{T}
    coefficient::T
    variable_1::VariableIndex
    variable_2::VariableIndex
end
```

Represents cx_ix_j where c is `coefficient`, x_i is the variable identified by `variable_1` and x_j is the variable identified by `variable_2`.

source

`MathOptInterface.ScalarQuadraticFunction` – Type.

```
ScalarQuadraticFunction{T}(quadratic_terms, affine_terms, constant)
```

The scalar-valued quadratic function $\frac{1}{2}x^TQx + a^Tx + b$, where:

- a is a sparse vector specified by a list of `ScalarAffineTerm` structs.
- b is a scalar specified by `constant`.
- Q is a symmetric matrix specified by a list of `ScalarQuadraticTerm` structs.

Duplicate indices in a or Q are accepted, and the corresponding coefficients are summed together. "Mirrored" indices `(q,r)` and `(r,q)` (where `r` and `q` are `VariableIndexes`) are considered duplicates; only one need be specified.

For example, for two scalar variables y, z, the quadratic expression $yz + y^2$ is represented by the terms `ScalarQuadraticTerm.([1.0, 2.0], [y, y], [z, y])`.

source

`MathOptInterface.VectorQuadraticTerm` – Type.

```
struct VectorQuadraticTerm{T}
    output_index::Int64
    scalar_term::ScalarQuadraticTerm{T}
end
```

A `ScalarQuadraticTerm` plus its index of the output component of a `VectorQuadraticFunction`. Each output component corresponds to a distinct sparse matrix Q_i.

source

`MathOptInterface.VectorQuadraticFunction` – Type.

```
VectorQuadraticFunction{T}(quadratic_terms, affine_terms, constants)
```

The vector-valued quadratic function with `i`th component ("output index") defined as $\frac{1}{2}x^TQ_ix + a_i^Tx + b_i$, where:

- a_i is a sparse vector specified by the `VectorAffineTerms` with `output_index == i`.
- b_i is a scalar specified by `constants[i]`
- Q_i is a symmetric matrix specified by the `VectorQuadraticTerm` with `output_index == i`.

Duplicate indices in a_i or Q_i are accepted, and the corresponding coefficients are summed together. "Mirrored" indices `(q,r)` and `(r,q)` (where `r` and `q` are `VariableIndexes`) are considered duplicates; only one need be specified.

source

Utilities

MathOptInterface.output_dimension – Function.

```
output_dimension(f::AbstractFunction)
```

Return 1 if f has a scalar output and the number of output components if f has a vector output.

source

MathOptInterface.constant – Method.

```
constant(f::Union{ScalarAffineFunction, ScalarQuadraticFunction})
```

Returns the constant term of the scalar function

source

MathOptInterface.constant – Method.

```
constant(f::Union{VectorAffineFunction, VectorQuadraticFunction})
```

Returns the vector of constant terms of the vector function

source

MathOptInterface.constant – Method.

```
constant(f::VariableIndex, ::Type{T}) where {T}
```

The constant term of a `VariableIndex` function is the zero value of the specified type T.

source

MathOptInterface.constant – Method.

```
constant(f::VectorOfVariables, ::Type{T}) where {T}
```

The constant term of a `VectorOfVariables` function is a vector of zero values of the specified type T.

source

18.2 Sets

MathOptInterface.AbstractSet – Type.

```
AbstractSet
```

Abstract supertype for set objects used to encode constraints. A set object should not contain any VariableIndex or ConstraintIndex as the set is passed unmodifed during copy_to.

source

MathOptInterface.AbstractScalarSet – Type.

```
AbstractScalarSet
```

Abstract supertype for subsets of $\mathbb{R}$.

source

MathOptInterface.AbstractVectorSet – Type.

```
AbstractVectorSet
```

Abstract supertype for subsets of $\mathbb{R}^n$ for some n.

source

Utilities

MathOptInterface.dimension – Function.

```
dimension(s::AbstractSet)
```

Return the output_dimension that an AbstractFunction should have to be used with the set s.

Examples

```
julia> dimension(Reals(4))
4

julia> dimension(LessThan(3.0))
1

julia> dimension(PositiveSemidefiniteConeTriangle(2))
3
```

source

MathOptInterface.dual_set – Function.

```
dual_set(s::AbstractSet)
```

Return the dual set of s, that is the dual cone of the set. This follows the definition of duality discussed in Duality.

See Dual cone for more information.

If the dual cone is not defined it returns an error.

Examples

```
julia> dual_set(Reals(4))
Zeros(4)

julia> dual_set(SecondOrderCone(5))
SecondOrderCone(5)

julia> dual_set(ExponentialCone())
DualExponentialCone()
```

source

MathOptInterface.dual_set_type – Function.

```
dual_set_type(S::Type{<:AbstractSet})
```

Return the type of dual set of sets of type S, as returned by dual_set. If the dual cone is not defined it returns an error.

Examples

```
julia> dual_set_type(Reals)
Zeros

julia> dual_set_type(SecondOrderCone)
SecondOrderCone

julia> dual_set_type(ExponentialCone)
DualExponentialCone
```

source

`MathOptInterface.constant` – Method.

```
constant(s::Union{EqualTo, GreaterThan, LessThan})
```

Returns the constant of the set.

source

`MathOptInterface.supports_dimension_update` – Function.

```
supports_dimension_update(S::Type{<:MOI.AbstractVectorSet})
```

Return a `Bool` indicating whether the elimination of any dimension of n-dimensional sets of type S give an `n-1`-dimensional set S. By default, this function returns `false` so it should only be implemented for sets that supports dimension update.

For instance, `supports_dimension_update(MOI.Nonnegatives}` is `true` because the elimination of any dimension of the n-dimensional nonnegative orthant gives the `n-1`-dimensional nonnegative orthant. However `supports_dimension_update(MOI.ExponentialCone}` is `false`.

source

`MathOptInterface.update_dimension` – Function.

```
update_dimension(s::AbstractVectorSet, new_dim)
```

Returns a set with the dimension modified to `new_dim`.

source

18.3 Scalar sets

List of recognized scalar sets.

`MathOptInterface.GreaterThan` – Type.

```
GreaterThan{T <: Real}(lower::T)
```

The set $[lower, \infty) \subseteq \mathbb{R}$.

source

`MathOptInterface.LessThan` – Type.

```
LessThan{T <: Real}(upper::T)
```

The set $(-\infty, upper] \subseteq \mathbb{R}$.

source

MathOptInterface.EqualTo – Type.

```
EqualTo{T <: Number}(value::T)
```

The set containing the single point $x \in \mathbb{R}$ where x is given by `value`.

source

MathOptInterface.Interval – Type.

```
Interval{T <: Real}(lower::T,upper::T)
```

The interval $[lower, upper] \subseteq \mathbb{R}$. If `lower` or `upper` is `-Inf` or `Inf`, respectively, the set is interpreted as a one-sided interval.

```
Interval(s::GreaterThan{<:AbstractFloat})
```

Construct a (right-unbounded) `Interval` equivalent to the given GreaterThan set.

```
Interval(s::LessThan{<:AbstractFloat})
```

Construct a (left-unbounded) `Interval` equivalent to the given LessThan set.

```
Interval(s::EqualTo{<:Real})
```

Construct a (degenerate) `Interval` equivalent to the given EqualTo set.

source

MathOptInterface.Integer – Type.

```
Integer()
```

The set of integers $\mathbb{Z}$.

source

MathOptInterface.ZeroOne – Type.

```
ZeroOne()
```

The set $\{0, 1\}$.

source

MathOptInterface.Semicontinuous – Type.

```
Semicontinuous{T <: Real}(lower::T,upper::T)
```

The set $\{0\} \cup [lower, upper]$.

source

MathOptInterface.Semiinteger – Type.

```
Semiinteger{T <: Real}(lower::T,upper::T)
```

The set $\{0\} \cup \{lower, lower + 1, \ldots, upper - 1, upper\}$.

source

18.4 Vector sets

List of recognized vector sets.

`MathOptInterface.Reals` – Type.

```
Reals(dimension)
```

The set $\mathbb{R}^{dimension}$ (containing all points) of dimension `dimension`.

source

`MathOptInterface.Zeros` – Type.

```
Zeros(dimension)
```

The set $\{0\}^{dimension}$ (containing only the origin) of dimension `dimension`.

source

`MathOptInterface.Nonnegatives` – Type.

```
Nonnegatives(dimension)
```

The nonnegative orthant $\{x \in \mathbb{R}^{dimension} : x \ge 0\}$ of dimension `dimension`.

source

`MathOptInterface.Nonpositives` – Type.

```
Nonpositives(dimension)
```

The nonpositive orthant $\{x \in \mathbb{R}^{dimension} : x \le 0\}$ of dimension `dimension`.

source

`MathOptInterface.NormInfinityCone` – Type.

```
NormInfinityCone(dimension)
```

The ℓ_∞-norm cone $\{(t,x) \in \mathbb{R}^{dimension} : t \ge \|x\|_\infty = \max_i |x_i|\}$ of dimension `dimension`.

source

`MathOptInterface.NormOneCone` – Type.

```
NormOneCone(dimension)
```

The ℓ_1-norm cone $\{(t,x) \in \mathbb{R}^{dimension} : t \ge \|x\|_1 = \sum_i |x_i|\}$ of dimension `dimension`.

source

`MathOptInterface.SecondOrderCone` – Type.

```
SecondOrderCone(dimension)
```

The second-order cone (or Lorenz cone or ℓ_2-norm cone) $\{(t,x) \in \mathbb{R}^{dimension} : t \ge \|x\|_2\}$ of dimension `dimension`.

source

MathOptInterface.RotatedSecondOrderCone – Type.

```
RotatedSecondOrderCone(dimension)
```

The rotated second-order cone $\{(t,u,x) \in \mathbb{R}^{dimension} : 2tu \ge \|x\|_2^2, t, u \ge 0\}$ of dimension dimension.

source

MathOptInterface.GeometricMeanCone – Type.

```
GeometricMeanCone(dimension)
```

The geometric mean cone $\{(t,x) \in \mathbb{R}^{n+1} : x \ge 0, t \le \sqrt[n]{x_1 x_2 \cdots x_n}\}$, where dimension = n + 1 >= 2.

Duality note

The dual of the geometric mean cone is $\{(u,v) \in \mathbb{R}^{n+1} : u \le 0, v \ge 0, -u \le n \sqrt[n]{\prod_i v_i}\}$, where dimension = n + 1 >= 2.

source

MathOptInterface.ExponentialCone – Type.

```
ExponentialCone()
```

The 3-dimensional exponential cone $\{(x,y,z) \in \mathbb{R}^3 : y \exp(x/y) \le z, y > 0\}$.

source

MathOptInterface.DualExponentialCone – Type.

```
DualExponentialCone()
```

The 3-dimensional dual exponential cone $\{(u,v,w) \in \mathbb{R}^3 : -u \exp(v/u) \le \exp(1) w, u < 0\}$.

source

MathOptInterface.PowerCone – Type.

```
PowerCone{T <: Real}(exponent::T)
```

The 3-dimensional power cone $\{(x,y,z) \in \mathbb{R}^3 : x^{exponent} y^{1-exponent} \ge |z|, x \ge 0, y \ge 0\}$ with parameter exponent.

source

MathOptInterface.DualPowerCone – Type.

```
DualPowerCone{T <: Real}(exponent::T)
```

The 3-dimensional power cone $\{(u,v,w) \in \mathbb{R}^3 : (\frac{u}{exponent})^{exponent} (\frac{v}{1-exponent})^{1-exponent} \ge |w|, u \ge 0, v \ge 0\}$ with parameter exponent.

source

MathOptInterface.RelativeEntropyCone – Type.

```
RelativeEntropyCone(dimension)
```

The relative entropy cone $\{(u, v, w) \in \mathbb{R}^{1+2n} : u \geq \sum_{i=1}^n w_i \log(\frac{w_i}{v_i}), v_i \geq 0, w_i \geq 0\}$, where `dimension = 2n + 1 >= 1`.

Duality note

The dual of the relative entropy cone is $\{(u, v, w) \in \mathbb{R}^{1+2n} : \forall i, w_i \geq u(\log(\frac{u}{v_i}) - 1), v_i \geq 0, u > 0\}$ of dimension `dimension` $= 2n + 1$.

source

`MathOptInterface.NormSpectralCone` – Type.

```
NormSpectralCone(row_dim, column_dim)
```

The epigraph of the matrix spectral norm (maximum singular value function) $\{(t, X) \in \mathbb{R}^{1+row_dim \times column_dim} : t \geq \sigma_1(X)\}$, where σ_i is the ith singular value of the matrix X of row dimension `row_dim` and column dimension `column_dim`.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

source

`MathOptInterface.NormNuclearCone` – Type.

```
NormNuclearCone(row_dim, column_dim)
```

The epigraph of the matrix nuclear norm (sum of singular values function) $\{(t, X) \in \mathbb{R}^{1+row_dim \times column_dim} : t \geq \sum_i \sigma_i(X)\}$, where σ_i is the ith singular value of the matrix X of row dimension `row_dim` and column dimension `column_dim`.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

source

`MathOptInterface.SOS1` – Type.

```
SOS1{T <: Real}(weights::Vector{T})
```

The set corresponding to the special ordered set (SOS) constraint of type 1. Of the variables in the set, at most one can be nonzero. The `weights` induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses.

source

`MathOptInterface.SOS2` – Type.

```
SOS2{T <: Real}(weights::Vector{T})
```

The set corresponding to the special ordered set (SOS) constraint of type 2. Of the variables in the set, at most two can be nonzero, and if two are nonzero, they must be adjacent in the ordering of the set. The `weights` induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses.

source

`MathOptInterface.Indicator` – Type.

```
Indicator{A<:ActivationCondition,S<:AbstractScalarSet}(set::S)
```

The set corresponding to an indicator constraint.

When A is ACTIVATE_ON_ZERO, this means: $\{(y, x) \in \{0, 1\} \times \mathbb{R}^n : y = 0 \implies x \in set\}$

When A is ACTIVATE_ON_ONE, this means: $\{(y, x) \in \{0, 1\} \times \mathbb{R}^n : y = 1 \implies x \in set\}$

Notes

Most solvers expect that the first row of the function is interpretable as a variable index x_i (e.g., `1.0 * x + 0.0`). An error will be thrown if this is not the case.

Example

The constraint $\{(y, x) \in \{0, 1\} \times \mathbb{R}^2 : y = 1 \implies x_1 + x_2 \leq 9\}$ is defined as

```
f = MOI.VectorAffineFunction(
    [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, y)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x1)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x2)),
    ],
    [0.0, 0.0],
)
s = MOI.Indicator{MOI.ACTIVATE_ON_ONE}(MOI.LessThan(9.0))
MOI.add_constraint(model, f, s)
```

source

`MathOptInterface.Complements` – Type.

```
Complements(dimension::Base.Integer)
```

The set corresponding to a mixed complementarity constraint.

Complementarity constraints should be specified with an `AbstractVectorFunction`-in-`Complements(dimension)` constraint.

The dimension of the vector-valued function F must be `dimension`. This defines a complementarity constraint between the scalar function `F[i]` and the variable in `F[i + dimension/2]`. Thus, `F[i + dimension/2]` must be interpretable as a single variable x_i (e.g., `1.0 * x + 0.0`), and `dimension` must be even.

The mixed complementarity problem consists of finding x_i in the interval `[lb, ub]` (i.e., in the set `Interval(lb, ub)`), such that the following holds:

1. `F_i(x) == 0` if `lb_i < x_i < ub_i`
2. `F_i(x) >= 0` if `lb_i == x_i`
3. `F_i(x) <= 0` if `x_i == ub_i`

Classically, the bounding set for x_i is `Interval(0, Inf)`, which recovers: `0 <= F_i(x) ⊥ x_i >= 0`, where the ⊥ operator implies `F_i(x) * x_i = 0`.

Examples

The problem:

```
x -in- Interval(-1, 1)
[-4 * x - 3, x] -in- Complements(2)
```

defines the mixed complementarity problem where the following holds:

1. `-4 * x - 3 == 0` if `-1 < x < 1`
2. `-4 * x - 3 >= 0` if `x == -1`
3. `-4 * x - 3 <= 0` if `x == 1`

There are three solutions:

1. `x = -3/4` with `F(x) = 0`
2. `x = -1` with `F(x) = 1`
3. `x = 1` with `F(x) = -7`

The function F can also be defined in terms of single variables. For example, the problem:

```
[x_3, x_4] -in- Nonnegatives(2)
[x_1, x_2, x_3, x_4] -in- Complements(4)
```

defines the complementarity problem where `0 <= x_1 ⊥ x_3 >= 0` and `0 <= x_2 ⊥ x_4 >= 0`.

source

18.5 Matrix sets

Matrix sets are vectorized in order to be subtypes of `AbstractVectorSet`.

For sets of symmetric matrices, storing both the `(i, j)` and `(j, i)` elements is redundant. Use the `AbstractSymmetricMatrixSe` set to represent only the vectorization of the upper triangular part of the matrix.

When the matrix of expressions constrained to be in the set is not symmetric, and hence additional constraints are needed to force the equality of the `(i, j)` and `(j, i)` elements, use the `AbstractSymmetricMatrixSetSquare` set.

The `Bridges.Constraint.SquareBridge` can transform a set from the square form to the `triangular_form` by adding appropriate constraints if the `(i, j)` and `(j, i)` expressions are different.

`MathOptInterface.AbstractSymmetricMatrixSetTriangle` – Type.

```
abstract type AbstractSymmetricMatrixSetTriangle <: AbstractVectorSet end
```

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with `side_dimension` rows and columns. The entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row). A vectorized cone of `dimension` n corresponds to a square matrix with side dimension $\sqrt{1/4 + 2n} - 1/2$. (Because a $d \times d$ matrix has $d(d + 1)/2$ elements in the upper or lower triangle.)

Examples

The matrix

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{bmatrix}$$

has `side_dimension` 3 and vectorization $(1, 2, 3, 4, 5, 6)$.

Note

Two packed storage formats exist for symmetric matrices, the respective orders of the entries are:

- upper triangular column by column (or lower triangular row by row);
- lower triangular column by column (or upper triangular row by row).

The advantage of the first format is the mapping between the `(i, j)` matrix indices and the `k` index of the vectorized form. It is simpler and does not depend on the side dimension of the matrix. Indeed,

- the entry of matrix indices `(i, j)` has vectorized index `k = div((j - 1) * j, 2) + i` if $i \le j$ and `k = div((i - 1) * i, 2) + j` if $j \le i$;
- and the entry with vectorized index `k` has matrix indices `i = div(1 + isqrt(8k - 7), 2)` and `j = k - div((i - 1) * i, 2)` or `j = div(1 + isqrt(8k - 7), 2)` and `i = k - div((j - 1) * j, 2)`.

Duality note

The scalar product for the symmetric matrix in its vectorized form is the sum of the pairwise product of the diagonal entries plus twice the sum of the pairwise product of the upper diagonal entries; see [p. 634, 1]. This has important consequence for duality.

Consider for example the following problem (`PositiveSemidefiniteConeTriangle` is a subtype of `AbstractSymmetricMatrix`

$$\begin{aligned} \max_{x \in \mathbb{R}} \quad & x \\ \text{s.t.} \quad & (1, -x, 1) \in \text{PositiveSemidefiniteConeTriangle}(2). \end{aligned}$$

The dual is the following problem

$$\begin{aligned} \min_{x \in \mathbb{R}^3} \quad & y_1 + y_3 \\ \text{s.t.} \quad & 2y_2 = 1 \\ & y \in \text{PositiveSemidefiniteConeTriangle}(2). \end{aligned}$$

Why do we use $2y_2$ in the dual constraint instead of y_2 ? The reason is that $2y_2$ is the scalar product between y and the symmetric matrix whose vectorized form is $(0, 1, 0)$. Indeed, with our modified scalar products we have

$$\langle (0,1,0), (y_1, y_2, y_3) \rangle = \operatorname{trace} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \\ y_2 & y_3 \end{pmatrix} = 2y_2.$$

References

[1] Boyd, S. and Vandenberghe, L.. Convex optimization. Cambridge university press, 2004.

source

`MathOptInterface.AbstractSymmetricMatrixSetSquare` – Type.

```
abstract type AbstractSymmetricMatrixSetSquare <: AbstractVectorSet end
```

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with `side_dimension` rows and columns. The entries of the matrix are given column by column (or equivalently, row by row). The matrix is both constrained to be symmetric and to have its `triangular_form` belong to the corresponding set. That is, if the functions in entries (i,j) and (j,i) are different, then a constraint will be added to make sure that the entries are equal.

Examples

PositiveSemidefiniteConeSquare is a subtype of AbstractSymmetricMatrixSetSquare and constraining the matrix

$$\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$$

to be symmetric positive semidefinite can be achieved by constraining the vector $(1,-z,-y,0)$ (or $(1,-y,-z,0)$) to belong to the `PositiveSemidefiniteConeSquare(2)`. It both constrains $y = z$ and $(1,-y,0)$ (or $(1,-z,0)$) to be in `PositiveSemidefiniteConeTriangle(2)`, since `triangular_form(PositiveSemidefiniteConeSquare)` is `PositiveSemidefiniteConeTriangle`.

source

MathOptInterface.side_dimension – Function.

```
side_dimension(set::Union{AbstractSymmetricMatrixSetTriangle,
                          AbstractSymmetricMatrixSetSquare})
```

Side dimension of the matrices in `set`. By convention, it should be stored in the `side_dimension` field but if it is not the case for a subtype of AbstractSymmetricMatrixSetTriangle, the method should be implemented for this subtype.

source

MathOptInterface.triangular_form – Function.

```
triangular_form(S::Type{<:AbstractSymmetricMatrixSetSquare})
triangular_form(set::AbstractSymmetricMatrixSetSquare)
```

Return the AbstractSymmetricMatrixSetTriangle corresponding to the vectorization of the upper triangular part of matrices in the AbstractSymmetricMatrixSetSquare set.

source

List of recognized matrix sets.

MathOptInterface.PositiveSemidefiniteConeTriangle – Type.

```
PositiveSemidefiniteConeTriangle(side_dimension) <: AbstractSymmetricMatrixSetTriangle
```

The (vectorized) cone of symmetric positive semidefinite matrices, with `side_dimension` rows and columns.

See AbstractSymmetricMatrixSetTriangle for more details on the vectorized form.

source

MathOptInterface.PositiveSemidefiniteConeSquare – Type.

```
PositiveSemidefiniteConeSquare(side_dimension) <: AbstractSymmetricMatrixSetSquare
```

The cone of symmetric positive semidefinite matrices, with side length `side_dimension`.

See AbstractSymmetricMatrixSetSquare for more details on the vectorized form.

The entries of the matrix are given column by column (or equivalently, row by row).

The matrix is both constrained to be symmetric and to be positive semidefinite. That is, if the functions in entries (i, j) and (j, i) are different, then a constraint will be added to make sure that the entries are equal.

Examples

Constraining the matrix

$$\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$$

to be symmetric positive semidefinite can be achieved by constraining the vector $(1, -z, -y, 0)$ (or $(1, -y, -z, 0)$) to belong to the `PositiveSemidefiniteConeSquare(2)`.

It both constrains $y = z$ and $(1, -y, 0)$ (or $(1, -z, 0)$) to be in `PositiveSemidefiniteConeTriangle(2)`.

source

`MathOptInterface.LogDetConeTriangle` – Type.

```
LogDetConeTriangle(side_dimension)
```

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d(d+1)/2} : t \le u \log(\det(X/u)), u > 0\}$, where the matrix X is represented in the same symmetric packed format as in the `PositiveSemidefiniteConeTriangle`.

The argument `side_dimension` is the side dimension of the matrix X, i.e., its number of rows or columns.

source

`MathOptInterface.LogDetConeSquare` – Type.

```
LogDetConeSquare(side_dimension)
```

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d^2} : t \le u \log(\det(X/u)), X \text{ symmetric}, u > 0\}$, where the matrix X is represented in the same format as in the `PositiveSemidefiniteConeSquare`.

Similarly to `PositiveSemidefiniteConeSquare`, constraints are added to ensure that X is symmetric.

The argument `side_dimension` is the side dimension of the matrix X, i.e., its number of rows or columns.

source

`MathOptInterface.RootDetConeTriangle` – Type.

```
RootDetConeTriangle(side_dimension)
```

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d(d+1)/2} : t \le \det(X)^{1/d}\}$, where the matrix X is represented in the same symmetric packed format as in the `PositiveSemidefiniteConeTriangle`.

The argument `side_dimension` is the side dimension of the matrix X, i.e., its number of rows or columns.

source

`MathOptInterface.RootDetConeSquare` – Type.

```
RootDetConeSquare(side_dimension)
```

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d^2} : t \le \det(X)^{1/d}, X \text{ symmetric}\}$, where the matrix X is represented in the same format as `PositiveSemidefiniteConeSquare`.

Similarly to `PositiveSemidefiniteConeSquare`, constraints are added to ensure that X is symmetric.

The argument `side_dimension` is the side dimension of the matrix X, i.e., its number of rows or columns.

source

Chapter 19

Models

19.1 Attribute interface

MathOptInterface.is_set_by_optimize – Function.

```
is_set_by_optimize(::AnyAttribute)
```

Return a Bool indicating whether the value of the attribute is modified during an optimize! call, that is, the attribute is used to query the result of the optimization.

Important note when defining new attributes

This function returns false by default so it should be implemented for attributes that are modified by optimize!.

source

MathOptInterface.is_copyable – Function.

```
is_copyable(::AnyAttribute)
```

Return a Bool indicating whether the value of the attribute may be copied during copy_to using set.

Important note when defining new attributes

By default is_copyable(attr) returns !is_set_by_optimize(attr). A specific method should be defined for attributes which are copied indirectly during copy_to. For instance, both is_copyable and is_set_by_optimize return false for the following attributes:

- ListOfOptimizerAttributesSet, ListOfModelAttributesSet, ListOfConstraintAttributesSet and ListOfVariableAttributesSet.
- SolverName and RawSolver: these attributes cannot be set.
- NumberOfVariables and ListOfVariableIndices: these attributes are set indirectly by add_variable and add_variables.
- ObjectiveFunctionType: this attribute is set indirectly when setting the ObjectiveFunction attribute.
- NumberOfConstraints, ListOfConstraintIndices, ListOfConstraintTypesPresent, CanonicalConstraintFunctio ConstraintFunction and ConstraintSet: these attributes are set indirectly by add_constraint and add_constraints.

source

MathOptInterface.get – Function.

```
get(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute)
```

Return an attribute `attr` of the optimizer `optimizer`.

```
get(model::ModelLike, attr::AbstractModelAttribute)
```

Return an attribute `attr` of the model `model`.

```
get(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex)
```

If the attribute `attr` is set for the variable v in the model `model`, return its value, return `nothing` otherwise. If the attribute `attr` is not supported by `model` then an error should be thrown instead of returning `nothing`.

```
get(model::ModelLike, attr::AbstractVariableAttribute, v::Vector{VariableIndex})
```

Return a vector of attributes corresponding to each variable in the collection `v` in the model `model`.

```
get(model::ModelLike, attr::AbstractConstraintAttribute, c::ConstraintIndex)
```

If the attribute `attr` is set for the constraint c in the model `model`, return its value, return `nothing` otherwise. If the attribute `attr` is not supported by `model` then an error should be thrown instead of returning `nothing`.

```
get(model::ModelLike, attr::AbstractConstraintAttribute, c::Vector{ConstraintIndex{F,S}})
```

Return a vector of attributes corresponding to each constraint in the collection `c` in the model `model`.

```
get(model::ModelLike, ::Type{VariableIndex}, name::String)
```

If a variable with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. Errors if two variables have the same name.

```
get(model::ModelLike, ::Type{ConstraintIndex{F,S}}, name::String) where {F<:AbstractFunction,S<:
    AbstractSet}
```

If an F-in-S constraint with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. Errors if two constraints have the same name.

```
get(model::ModelLike, ::Type{ConstraintIndex}, name::String)
```

If any constraint with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. This version is available for convenience but may incur a performance penalty because it is not type stable. Errors if two constraints have the same name.

Examples

```
get(model, ObjectiveValue())
get(model, VariablePrimal(), ref)
get(model, VariablePrimal(5), [ref1, ref2])
get(model, OtherAttribute("something specific to cplex"))
get(model, VariableIndex, "var1")
get(model, ConstraintIndex{ScalarAffineFunction{Float64},LessThan{Float64}}, "con1")
get(model, ConstraintIndex, "con1")
```

source

MathOptInterface.get! – Function.

```
get!(output, model::ModelLike, args...)
```

An in-place version of get.

The signature matches that of get except that the the result is placed in the vector output.

source

MathOptInterface.set – Function.

```
set(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute, value)
```

Assign value to the attribute attr of the optimizer optimizer.

```
set(model::ModelLike, attr::AbstractModelAttribute, value)
```

Assign value to the attribute attr of the model model.

```
set(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex, value)
```

Assign value to the attribute attr of variable v in model model.

```
set(model::ModelLike, attr::AbstractVariableAttribute, v::Vector{VariableIndex}, vector_of_values
    )
```

Assign a value respectively to the attribute attr of each variable in the collection v in model model.

```
set(model::ModelLike, attr::AbstractConstraintAttribute, c::ConstraintIndex, value)
```

Assign a value to the attribute attr of constraint c in model model.

```
set(model::ModelLike, attr::AbstractConstraintAttribute, c::Vector{ConstraintIndex{F,S}},
    vector_of_values)
```

Assign a value respectively to the attribute attr of each constraint in the collection c in model model.

An UnsupportedAttribute error is thrown if model does not support the attribute attr (see supports) and a SetAttributeNotAllowed error is thrown if it supports the attribute attr but it cannot be set.

Replace set in a constraint

```
set(model::ModelLike, ::ConstraintSet, c::ConstraintIndex{F,S}, set::S)
```

Change the set of constraint c to the new set set which should be of the same type as the original set.

Examples

If c is a ConstraintIndex{F,Interval}

```
set(model, ConstraintSet(), c, Interval(0, 5))
set(model, ConstraintSet(), c, GreaterThan(0.0))  # Error
```

Replace function in a constraint

```
set(model::ModelLike, ::ConstraintFunction, c::ConstraintIndex{F,S}, func::F)
```

Replace the function in constraint c with func. F must match the original function type used to define the constraint.

Note

Setting the constraint function is not allowed if F is VariableIndex, it throws a SettingVariableIndexNotAllowed error. Indeed, it would require changing the index c as the index of VariableIndex constraints should be the same as the index of the variable.

Examples

If c is a ConstraintIndex{ScalarAffineFunction,S} and v1 and v2 are VariableIndex objects,

```
set(model, ConstraintFunction(), c,
    ScalarAffineFunction(ScalarAffineTerm.([1.0, 2.0], [v1, v2]), 5.0))
set(model, ConstraintFunction(), c, v1) # Error
```

source

MathOptInterface.supports – Function.

```
supports(model::ModelLike, sub::AbstractSubmittable)::Bool
```

Return a Bool indicating whether model supports the submittable sub.

```
supports(model::ModelLike, attr::AbstractOptimizerAttribute)::Bool
```

Return a Bool indicating whether model supports the optimizer attribute attr. That is, it returns false if copy_to(model, src) shows a warning in case attr is in the ListOfOptimizerAttributesSet of src; see copy_to for more details on how unsupported optimizer attributes are handled in copy.

```
supports(model::ModelLike, attr::AbstractModelAttribute)::Bool
```

Return a Bool indicating whether model supports the model attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfModelAttributesSet of src.

```
supports(model::ModelLike, attr::AbstractVariableAttribute, ::Type{VariableIndex})::Bool
```

Return a Bool indicating whether model supports the variable attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfVariableAttributesSet of src.

```
supports(model::ModelLike, attr::AbstractConstraintAttribute, ::Type{ConstraintIndex{F,S}})::Bool
     where {F,S}
```

Return a Bool indicating whether model supports the constraint attribute attr applied to an F-in-S constraint. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfConstraintAttributesSet of src.

For all five methods, if the attribute is only not supported in specific circumstances, it should still return true.

Note that supports is only defined for attributes for which is_copyable returns true as other attributes do not appear in the list of attributes set obtained by ListOf...AttributesSet.

source

MathOptInterface.attribute_value_type – Function.

```
attribute_value_type(attr::AnyAttribute)
```

Given an attribute attr, return the type of value expected by get, or returned by set.

Notes

- Only implement this if it make sense to do so. If un-implemented, the default is Any.

source

19.2 Model interface

MathOptInterface.ModelLike – Type.

```
ModelLike
```

Abstract supertype for objects that implement the "Model" interface for defining an optimization problem.

source

MathOptInterface.is_empty – Function.

```
is_empty(model::ModelLike)
```

Returns `false` if the `model` has any model attribute set or has any variables or constraints.

Note that an empty model can have optimizer attributes set.

source

MathOptInterface.empty! – Function.

```
empty!(model::ModelLike)
```

Empty the model, that is, remove all variables, constraints and model attributes but not optimizer attributes.

source

MathOptInterface.write_to_file – Function.

```
write_to_file(model::ModelLike, filename::String)
```

Writes the current model data to the given file. Supported file types depend on the model type.

source

MathOptInterface.read_from_file – Function.

```
read_from_file(model::ModelLike, filename::String)
```

Read the file `filename` into the model `model`. If `model` is non-empty, this may throw an error.

Supported file types depend on the model type.

Note

Once the contents of the file are loaded into the model, users can query the variables via `get(model, ListOfVariableIndices())`. However, some filetypes, such as LP files, do not maintain an explicit ordering of the variables. Therefore, the returned list may be in an arbitrary order. To avoid depending on the order of the indices, users should look up each variable index by name: `get(model, VariableIndex, "name")`.

source

MathOptInterface.supports_incremental_interface – Function.

```
supports_incremental_interface(model::ModelLike)
```

Return a `Bool` indicating whether `model` supports building incrementally via `add_variable` and `add_constraint`.

The main purpose of this function is to determine whether a model can be loaded into `model` incrementally or whether it should be cached and copied at once instead.

source

`MathOptInterface.copy_to` – Function.

```
copy_to(dest::ModelLike, src::ModelLike)::IndexMap
```

Copy the model from `src` into `dest`.

The target `dest` is emptied, and all previous indices to variables and constraints in `dest` are invalidated.

Returns an `IndexMap` object that translates variable and constraint indices from the `src` model to the corresponding indices in the `dest` model.

Notes

- If a constraint that in `src` is not supported by `dest`, then an `UnsupportedConstraint` error is thrown.
- If an `AbstractModelAttribute`, `AbstractVariableAttribute`, or `AbstractConstraintAttribute` is set in `src` but not supported by `dest`, then an `UnsupportedAttribute` error is thrown.

`AbstractOptimizerAttribute`s are not copied to the `dest` model.

IndexMap

Implementations of `copy_to` must return an `IndexMap`. For technical reasons, this type is defined in the Utilities submodule as `MOI.Utilities.IndexMap`. However, since it is an integral part of the MOI API, we provide `MOI.IndexMap` as an alias.

Example

```
# Given empty `ModelLike` objects `src` and `dest`.

x = add_variable(src)

is_valid(src, x)   # true
is_valid(dest, x)  # false (`dest` has no variables)

index_map = copy_to(dest, src)
is_valid(dest, x) # false (unless index_map[x] == x)
is_valid(dest, index_map[x]) # true
```

source

`MathOptInterface.IndexMap` – Type.

```
IndexMap()
```

The dictionary-like object returned by `copy_to`.

IndexMap

Implementations of `copy_to` must return an `IndexMap`. For technical reasons, the `IndexMap` type is defined in the Utilties submodule as `MOI.Utilities.IndexMap`. However, since it is an integral part of the MOI API, we provide this `MOI.IndexMap` as an alias.

source

19.3 Model attributes

MathOptInterface.AbstractModelAttribute – Type.

```
AbstractModelAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the model.

source

MathOptInterface.Name – Type.

```
Name()
```

A model attribute for the string identifying the model. It has a default value of `""` if not set'.

source

MathOptInterface.ObjectiveFunction – Type.

```
ObjectiveFunction{F<:AbstractScalarFunction}()
```

A model attribute for the objective function which has a type `F<:AbstractScalarFunction`. `F` should be guaranteed to be equivalent but not necessarily identical to the function type provided by the user. Throws an `InexactError` if the objective function cannot be converted to `F`, e.g. the objective function is quadratic and `F` is `ScalarAffineFunction{Float64}` or it has non-integer coefficient and `F` is `ScalarAffineFunction{Int}`.

source

MathOptInterface.ObjectiveFunctionType – Type.

```
ObjectiveFunctionType()
```

A model attribute for the type `F` of the objective function set using the `ObjectiveFunction{F}` attribute.

Examples

In the following code, `attr` should be equal to `MOI.VariableIndex`:

```
x = MOI.add_variable(model)
MOI.set(model, MOI.ObjectiveFunction{MOI.VariableIndex}(),
         x)
attr = MOI.get(model, MOI.ObjectiveFunctionType())
```

source

MathOptInterface.ObjectiveSense – Type.

```
ObjectiveSense()
```

A model attribute for the objective sense of the objective function, which must be an `OptimizationSense`: `MIN_SENSE`, `MAX_SENSE`, or `FEASIBILITY_SENSE`. The default is `FEASIBILITY_SENSE`.

source

MathOptInterface.NumberOfVariables – Type.

```
NumberOfVariables()
```

A model attribute for the number of variables in the model.

source

MathOptInterface.ListOfVariableIndices – Type.

```
ListOfVariableIndices()
```

A model attribute for the `Vector{VariableIndex}` of all variable indices present in the model (i.e., of length equal to the value of `NumberOfVariables()`) in the order in which they were added.

source

MathOptInterface.ListOfConstraintTypesPresent – Type.

```
ListOfConstraintTypesPresent()
```

A model attribute for the list of tuples of the form `(F,S)`, where F is a function type and S is a set type indicating that the attribute `NumberOfConstraints{F,S}()` has value greater than zero.

source

MathOptInterface.NumberOfConstraints – Type.

```
NumberOfConstraints{F,S}()
```

A model attribute for the number of constraints of the type F-in-S present in the model.

source

MathOptInterface.ListOfConstraintIndices – Type.

```
ListOfConstraintIndices{F,S}()
```

A model attribute for the `Vector{ConstraintIndex{F,S}}` of all constraint indices of type F-in-S in the model (i.e., of length equal to the value of `NumberOfConstraints{F,S}()`) in the order in which they were added.

source

MathOptInterface.ListOfOptimizerAttributesSet – Type.

```
ListOfOptimizerAttributesSet()
```

An optimizer attribute for the `Vector{AbstractOptimizerAttribute}` of all optimizer attributes that were set.

source

MathOptInterface.ListOfModelAttributesSet – Type.

```
ListOfModelAttributesSet()
```

A model attribute for the `Vector{AbstractModelAttribute}` of all model attributes `attr` such that 1) `is_copyable(attr)` returns `true` and 2) the attribute was set to the model.

source

MathOptInterface.ListOfVariableAttributesSet – Type.

```
ListOfVariableAttributesSet()
```

A model attribute for the `Vector{AbstractVariableAttribute}` of all variable attributes `attr` such that 1) `is_copyable(attr)` returns `true` and 2) the attribute was set to variables.

source

`MathOptInterface.ListOfConstraintAttributesSet` – Type.

```
ListOfConstraintAttributesSet{F, S}()
```

A model attribute for the `Vector{AbstractConstraintAttribute}` of all constraint attributes `attr` such that 1) `is_copyable(attr)` returns `true` and

2. the attribute was set to F-in-S constraints.

Note

The attributes `ConstraintFunction` and `ConstraintSet` should not be included in the list even if then have been set with `set`.

source

19.4 Optimizer interface

`MathOptInterface.AbstractOptimizer` – Type.

```
AbstractOptimizer <: ModelLike
```

Abstract supertype for objects representing an instance of an optimization problem tied to a particular solver. This is typically a solver's in-memory representation. In addition to `ModelLike`, `AbstractOptimizer` objects let you solve the model and query the solution.

source

`MathOptInterface.OptimizerWithAttributes` – Type.

```
struct OptimizerWithAttributes
    optimizer_constructor
    params::Vector{Pair{AbstractOptimizerAttribute,<:Any}}
end
```

Object grouping an optimizer constructor and a list of optimizer attributes. Instances are created with `instantiate`.

source

`MathOptInterface.optimize!` – Function.

```
optimize!(optimizer::AbstractOptimizer)
```

Optimize the problem contained in `optimizer`.

Before calling `optimize!`, the problem should first be constructed using the incremental interface (see `supports_incremental_interface`) or `copy_to`.

source

`MathOptInterface.instantiate` – Function.

```
instantiate(
    optimizer_constructor,
    with_bridge_type::Union{Nothing, Type} = nothing,
)
```

Creates an instance of optimizer by either:

- calling `optimizer_constructor.optimizer_constructor()` and setting the parameters in `optimizer_constructor.p` if `optimizer_constructor` is a `OptimizerWithAttributes`
- calling `optimizer_constructor()` if `optimizer_constructor` is callable.

If `with_bridge_type` is not `nothing`, it enables all the bridges defined in the MathOptInterface.Bridges submodule with coefficient type `with_bridge_type`.

If the optimizer created by `optimizer_constructor` does not support loading the problem incrementally (see `supports_incremental_interface`), then a `Utilities.CachingOptimizer` is added to store a cache of the bridged model.

source

19.5 Optimizer attributes

`MathOptInterface.AbstractOptimizerAttribute` – Type.

```
AbstractOptimizerAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the optimizer.

Note

The difference between `AbstractOptimizerAttribute` and `AbstractModelAttribute` lies in the behavior of `is_empty`, `empty!` and `copy_to`. Typically optimizer attributes only affect how the model is solved.

source

`MathOptInterface.SolverName` – Type.

```
SolverName()
```

An optimizer attribute for the string identifying the solver/optimizer.

source

`MathOptInterface.SolverVersion` – Type.

```
SolverVersion()
```

An optimizer attribute for the string identifying the version of the solver.

> **Note**
>
> For solvers supporting semantic versioning, the `SolverVersion` should be a string of the form "vMAJOR.MINOR.PATCH", so that it can be converted to a Julia `VersionNumber` (e.g., 'VersionNumber("v1.2.3")).
>
> We do not require Semantic Versioning because some solvers use alternate versioning systems. For example, CPLEX uses Calendar Versioning, so `SolverVersion` will return a string like `"202001"`.

source

`MathOptInterface.Silent` – Type.

```
Silent()
```

An optimizer attribute for silencing the output of an optimizer. When `set` to `true`, it takes precedence over any other attribute controlling verbosity and requires the solver to produce no output. The default value is `false` which has no effect. In this case the verbosity is controlled by other attributes.

Note

Every optimizer should have verbosity on by default. For instance, if a solver has a solver-specific log level attribute, the MOI implementation should set it to `1` by default. If the user sets `Silent` to `true`, then the log level should be set to `0`, even if the user specifically sets a value of log level. If the value of `Silent` is `false` then the log level set to the solver is the value given by the user for this solver-specific parameter or `1` if none is given.

source

`MathOptInterface.TimeLimitSec` – Type.

```
TimeLimitSec()
```

An optimizer attribute for setting a time limit for an optimization. When `set` to `nothing`, it deactivates the solver time limit. The default value is `nothing`. The time limit is in seconds.

source

`MathOptInterface.RawOptimizerAttribute` – Type.

```
RawOptimizerAttribute(name::String)
```

An optimizer attribute for the solver-specific parameter identified by `name`.

source

`MathOptInterface.NumberOfThreads` – Type.

```
NumberOfThreads()
```

An optimizer attribute for setting the number of threads used for an optimization. When set to `nothing` uses solver default. Values are positive integers. The default value is `nothing`.

source

`MathOptInterface.RawSolver` – Type.

```
RawSolver()
```

A model attribute for the object that may be used to access a solver-specific API for this optimizer.

source

List of attributes useful for optimizers

`MathOptInterface.TerminationStatus` – Type.

```
TerminationStatus()
```

A model attribute for the `TerminationStatusCode` explaining why the optimizer stopped.

source

`MathOptInterface.TerminationStatusCode` – Type.

```
TerminationStatusCode
```

An Enum of possible values for the `TerminationStatus` attribute. This attribute is meant to explain the reason why the optimizer stopped executing in the most recent call to `optimize!`.

If no call has been made to `optimize!`, then the `TerminationStatus` is:

- `OPTIMIZE_NOT_CALLED`: The algorithm has not started.

OK

These are generally OK statuses, i.e., the algorithm ran to completion normally.

- `OPTIMAL`: The algorithm found a globally optimal solution.
- `INFEASIBLE`: The algorithm concluded that no feasible solution exists.
- `DUAL_INFEASIBLE`: The algorithm concluded that no dual bound exists for the problem. If, additionally, a feasible (primal) solution is known to exist, this status typically implies that the problem is unbounded, with some technical exceptions.
- `LOCALLY_SOLVED`: The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.
- `LOCALLY_INFEASIBLE`: The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.
- `INFEASIBLE_OR_UNBOUNDED`: The algorithm stopped because it decided that the problem is infeasible or unbounded; this occasionally happens during MIP presolve.

Solved to relaxed tolerances

- `ALMOST_OPTIMAL`: The algorithm found a globally optimal solution to relaxed tolerances.
- `ALMOST_INFEASIBLE`: The algorithm concluded that no feasible solution exists within relaxed tolerances.
- `ALMOST_DUAL_INFEASIBLE`: The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.
- `ALMOST_LOCALLY_SOLVED`: The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

Limits

The optimizer stopped because of some user-defined limit.

- `ITERATION_LIMIT`: An iterative algorithm stopped after conducting the maximum number of iterations.
- `TIME_LIMIT`: The algorithm stopped after a user-specified computation time.
- `NODE_LIMIT`: A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branch-and-bound tree.

- SOLUTION_LIMIT: The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.
- MEMORY_LIMIT: The algorithm stopped because it ran out of memory.
- OBJECTIVE_LIMIT: The algorithm stopped because it found a solution better than a minimum limit set by the user.
- NORM_LIMIT: The algorithm stopped because the norm of an iterate became too large.
- OTHER_LIMIT: The algorithm stopped due to a limit not covered by one of the above.

Problematic

This group of statuses means that something unexpected or problematic happened.

- SLOW_PROGRESS: The algorithm stopped because it was unable to continue making progress towards the solution.
- NUMERICAL_ERROR: The algorithm stopped because it encountered unrecoverable numerical error.
- INVALID_MODEL: The algorithm stopped because the model is invalid.
- INVALID_OPTION: The algorithm stopped because it was provided an invalid option.
- INTERRUPTED: The algorithm stopped because of an interrupt signal.
- OTHER_ERROR: The algorithm stopped because of an error not covered by one of the statuses defined above.

source

MathOptInterface.PrimalStatus – Type.

```
PrimalStatus(result_index::Int = 1)
```

A model attribute for the ResultStatusCode of the primal result result_index. If result_index is omitted, it defaults to 1.

See ResultCount for information on how the results are ordered.

If result_index is larger than the value of ResultCount then NO_SOLUTION is returned.

source

MathOptInterface.DualStatus – Type.

```
DualStatus(result_index::Int = 1)
```

A model attribute for the ResultStatusCode of the dual result result_index. If result_index is omitted, it defaults to 1.

See ResultCount for information on how the results are ordered.

If result_index is larger than the value of ResultCount then NO_SOLUTION is returned.

source

MathOptInterface.ResultStatusCode – Type.

```
ResultStatusCode
```

An Enum of possible values for the PrimalStatus and DualStatus attributes. The values indicate how to interpret the result vector.

- NO_SOLUTION: the result vector is empty.
- FEASIBLE_POINT: the result vector is a feasible point.
- NEARLY_FEASIBLE_POINT: the result vector is feasible if some constraint tolerances are relaxed.
- INFEASIBLE_POINT: the result vector is an infeasible point.
- INFEASIBILITY_CERTIFICATE: the result vector is an infeasibility certificate. If the PrimalStatus is INFEASIBILITY_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility. If the DualStatus is INFEASIBILITY_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.
- NEARLY_INFEASIBILITY_CERTIFICATE: the result satisfies a relaxed criterion for a certificate of infeasibility.
- REDUCTION_CERTIFICATE: the result vector is an ill-posed certificate; see this article for details. If the PrimalStatus is REDUCTION_CERTIFICATE, then the primal result vector is a proof that the dual problem is ill-posed. If the DualStatus is REDUCTION_CERTIFICATE, then the dual result vector is a proof that the primal is ill-posed.
- NEARLY_REDUCTION_CERTIFICATE: the result satisfies a relaxed criterion for an ill-posed certificate.
- UNKNOWN_RESULT_STATUS: the result vector contains a solution with an unknown interpretation.
- OTHER_RESULT_STATUS: the result vector contains a solution with an interpretation not covered by one of the statuses defined above.

source

MathOptInterface.RawStatusString – Type.

```
RawStatusString()
```

A model attribute for a solver specific string explaining why the optimizer stopped.

source

MathOptInterface.ResultCount – Type.

```
ResultCount()
```

A model attribute for the number of results available.

Order of solutions

A number of attributes contain an index, `result_index`, which is used to refer to one of the available results. Thus, `result_index` must be an integer between 1 and the number of available results.

As a general rule, the first result (`result_index=1`) is the most important result (e.g., an optimal solution or an infeasibility certificate). Other results will typically be alternate solutions that the solver found during the search for the first result.

If a (local) optimal solution is available, i.e., TerminationStatus is OPTIMAL or LOCALLY_SOLVED, the first result must correspond to the (locally) optimal solution. Other results may be alternative optimal solutions, or they may be other suboptimal solutions; use ObjectiveValue to distingiush between them.

If a primal or dual infeasibility certificate is available, i.e., TerminationStatus is INFEASIBLE or DUAL_INFEASIBLE and the corresponding PrimalStatus or DualStatus is INFEASIBILITY_CERTIFICATE, then the first result must be a certificate. Other results may be alternate certificates, or infeasible points.

source

MathOptInterface.ObjectiveValue – Type.

```
ObjectiveValue(result_index::Int = 1)
```

A model attribute for the objective value of the primal solution `result_index`.

If the solver does not have a primal value for the objective because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `ObjectiveValue` attribute.

See `ResultCount` for information on how the results are ordered.

source

`MathOptInterface.DualObjectiveValue` – Type.

```
DualObjectiveValue(result_index::Int = 1)
```

A model attribute for the value of the objective function of the dual problem for the `result_index`th dual result.

If the solver does not have a dual value for the objective because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a primal solution is available), the result is undefined. Users should first check `DualStatus` before accessing the `DualObjectiveValue` attribute.

See `ResultCount` for information on how the results are ordered.

source

`MathOptInterface.ObjectiveBound` – Type.

```
ObjectiveBound()
```

A model attribute for the best known bound on the optimal objective value.

source

`MathOptInterface.RelativeGap` – Type.

```
RelativeGap()
```

A model attribute for the final relative optimality gap.

Warning

The definition of this gap is solver-dependent. However, most solvers implementing this attribute define the relative gap as some variation of $\frac{|b-f|}{|f|}$, where b is the best bound and f is the best feasible objective value.

source

`MathOptInterface.SolveTimeSec` – Type.

```
SolveTimeSec()
```

A model attribute for the total elapsed solution time (in seconds) as reported by the optimizer.

source

MathOptInterface.SimplexIterations – Type.

```
SimplexIterations()
```

A model attribute for the cumulative number of simplex iterations during the optimization process. In particular, for a mixed-integer program (MIP), the total simplex iterations for all nodes.

source

MathOptInterface.BarrierIterations – Type.

```
BarrierIterations()
```

A model attribute for the cumulative number of barrier iterations while solving a problem.

source

MathOptInterface.NodeCount – Type.

```
NodeCount()
```

A model attribute for the total number of branch-and-bound nodes explored while solving a mixed-integer program (MIP).

source

Conflict Status

MathOptInterface.compute_conflict! – Function.

```
compute_conflict!(optimizer::AbstractOptimizer)
```

Computes a minimal subset of constraints such that the model with the other constraint removed is still infeasible.

Some solvers call a set of conflicting constraints an Irreducible Inconsistent Subsystem (IIS).

See also ConflictStatus and ConstraintConflictStatus.

Note

If the model is modified after a call to `compute_conflict!`, the implementor is not obliged to purge the conflict. Any calls to the above attributes may return values for the original conflict without a warning. Similarly, when modifying the model, the conflict can be discarded.

source

MathOptInterface.ConflictStatus – Type.

```
ConflictStatus()
```

A model attribute for the ConflictStatusCode explaining why the conflict refiner stopped when computing the conflict.

source

MathOptInterface.ConflictStatusCode – Type.

```
ConflictStatusCode
```

An Enum of possible values for the `ConflictStatus` attribute. This attribute is meant to explain the reason why the conflict finder stopped executing in the most recent call to `compute_conflict!`.

Possible values are:

- `COMPUTE_CONFLICT_NOT_CALLED`: the function `compute_conflict!` has not yet been called
- `NO_CONFLICT_EXISTS`: there is no conflict because the problem is feasible
- `NO_CONFLICT_FOUND`: the solver could not find a conflict
- `CONFLICT_FOUND`: at least one conflict could be found

source

`MathOptInterface.ConstraintConflictStatus` – Type.

```
ConstraintConflictStatus()
```

A constraint attribute indicating whether the constraint participates in the conflict. Its type is `ConflictParticipationStatus(`

source

`MathOptInterface.ConflictParticipationStatusCode` – Type.

```
ConflictParticipationStatusCode
```

An Enum of possible values for the `ConstraintConflictStatus` attribute. This attribute is meant to indicate whether a given constraint participates or not in the last computed conflict.

Possible values are:

- `NOT_IN_CONFLICT`: the constraint does not participate in the conflict
- `IN_CONFLICT`: the constraint participates in the conflict
- `MAYBE_IN_CONFLICT`: the constraint may participate in the conflict, the solver was not able to prove that the constraint can be excluded from the conflict

source

Chapter 20

Variables

20.1 Functions

MathOptInterface.add_variable – Function.

```
add_variable(model::ModelLike)::VariableIndex
```

Add a scalar variable to the model, returning a variable index.

A AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

source

MathOptInterface.add_variables – Function.

```
add_variables(model::ModelLike, n::Int)::Vector{VariableIndex}
```

Add n scalar variables to the model, returning a vector of variable indices.

A AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

source

MathOptInterface.add_constrained_variable – Function.

```
add_constrained_variable(
    model::ModelLike,
    set::AbstractScalarSet
)::Tuple{MOI.VariableIndex,
         MOI.ConstraintIndex{MOI.VariableIndex, typeof(set)}}
```

Add to model a scalar variable constrained to belong to set, returning the index of the variable created and the index of the constraint constraining the variable to belong to set.

By default, this function falls back to creating a free variable with add_variable and then constraining it to belong to set with add_constraint.

source

MathOptInterface.add_constrained_variables – Function.

```
add_constrained_variables(
    model::ModelLike,
    sets::AbstractVector{<:AbstractScalarSet}
)::Tuple{
    Vector{MOI.VariableIndex},
    Vector{MOI.ConstraintIndex{MOI.VariableIndex,eltype(sets)}},
}
```

Add to `model` scalar variables constrained to belong to `sets`, returning the indices of the variables created and the indices of the constraints constraining the variables to belong to each set in `sets`. That is, if it returns `variables` and `constraints`, `constraints[i]` is the index of the constraint constraining `variable[i]` to belong to `sets[i]`.

By default, this function falls back to calling `add_constrained_variable` on each set.

source

```
add_constrained_variables(
    model::ModelLike,
    set::AbstractVectorSet,
)::Tuple{
    Vector{MOI.VariableIndex},
    MOI.ConstraintIndex{MOI.VectorOfVariables,typeof(set)},
}
```

Add to `model` a vector of variables constrained to belong to `set`, returning the indices of the variables created and the index of the constraint constraining the vector of variables to belong to `set`.

By default, this function falls back to creating free variables with `add_variables` and then constraining it to belong to `set` with `add_constraint`.

source

`MathOptInterface.supports_add_constrained_variable` – Function.

```
supports_add_constrained_variable(
    model::ModelLike,
    S::Type{<:AbstractScalarSet}
)::Bool
```

Return a `Bool` indicating whether `model` supports constraining a variable to belong to a set of type `S` either on creation of the variable with `add_constrained_variable` or after the variable is created with `add_constraint`.

By default, this function falls back to `supports_add_constrained_variables(model, Reals) && supports_constraint(mo` `MOI.VariableIndex, S)` which is the correct definition for most models.

Example

Suppose that a solver supports only two kind of variables: binary variables and continuous variables with a lower bound. If the solver decides not to support `VariableIndex`-in-`Binary` and `VariableIndex`-in-`GreaterThan` constraints, it only has to implement `add_constrained_variable` for these two sets which prevents the user to add both a binary constraint and a lower bound on the same variable. Moreover, if the user adds a `VariableIndex`-in-`GreaterThan` constraint, implementing this interface (i.e., `supports_add_constrained_varia` enables the constraint to be transparently bridged into a supported constraint.

source

`MathOptInterface.supports_add_constrained_variables` – Function.

```
supports_add_constrained_variables(
    model::ModelLike,
    S::Type{<:AbstractVectorSet}
)::Bool
```

Return a `Bool` indicating whether `model` supports constraining a vector of variables to belong to a set of type `S` either on creation of the vector of variables with `add_constrained_variables` or after the variable is created with `add_constraint`.

By default, if `S` is `Reals` then this function returns `true` and otherwise, it falls back to `supports_add_constrained_variables(Reals) && supports_constraint(model, MOI.VectorOfVariables, S)` which is the correct definition for most models.

Example

In the standard conic form (see Duality), the variables are grouped into several cones and the constraints are affine equality constraints. If `Reals` is not one of the cones supported by the solvers then it needs to implement `supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false` as free variables are not supported. The solvers should then implement `supports_add_constrained_variables(::Optimizer, ::Type{<:SupportedCones}) = true` where `SupportedCones` is the union of all cone types that are supported; it does not have to implement the method `supports_constraint(::Type{VectorOfVariables}, Type{<:SupportedCones})` as it should return `false` and it's the default. This prevents the user to constrain the same variable in two different cones. When a `VectorOfVariables`-in-`S` is added, the variables of the vector have already been created so they already belong to given cones. If bridges are enabled, the constraint will therefore be bridged by adding slack variables in `S` and equality constraints ensuring that the slack variables are equal to the corresponding variables of the given constraint function.

Note that there may also be sets for which `!supports_add_constrained_variables(model, S)` and `supports_constraint(model, MOI.VectorOfVariables, S)`. For instance, suppose a solver supports positive semidefinite variable constraints and two types of variables: binary variables and nonnegative variables. Then the solver should support adding `VectorOfVariables`-in-`PositiveSemidefiniteConeTriangle` constraints, but it should not support creating variables constrained to belong to the `PositiveSemidefiniteConeTriangle` because the variables in `PositiveSemidefiniteConeTriangle` should first be created as either binary or non-negative.

source

`MathOptInterface.is_valid` – Method.

```
is_valid(model::ModelLike, index::Index)::Bool
```

Return a `Bool` indicating whether this index refers to a valid object in the model `model`.

source

`MathOptInterface.delete` – Method.

```
delete(model::ModelLike, index::Index)
```

Delete the referenced object from the model. Throw `DeleteNotAllowed` if if `index` cannot be deleted.

The following modifications also take effect if `Index` is `VariableIndex`:

- If `index` used in the objective function, it is removed from the function, i.e., it is substituted for zero.
- For each `func`-in-`set` constraint of the model:
  - If `func isa VariableIndex` and `func == index` then the constraint is deleted.

- If `func isa VectorOfVariables` and `index in func.variables` then
  * if `length(func.variables) == 1` is one, the constraint is deleted;
  * if `length(func.variables) > 1` and `supports_dimension_update(set)` then then the variable is removed from `func` and `set` is replaced by `update_dimension(set, MOI.dimension(set) - 1)`.
  * Otherwise, a `DeleteNotAllowed` error is thrown.
- Otherwise, the variable is removed from `func`, i.e., it is substituted for zero.

source

`MathOptInterface.delete` – Method.

```
delete(model::ModelLike, indices::Vector{R<:Index}) where {R}
```

Delete the referenced objects in the vector `indices` from the model. It may be assumed that R is a concrete type. The default fallback sequentially deletes the individual items in `indices`, although specialized implementations may be more efficient.

source

20.2 Attributes

`MathOptInterface.AbstractVariableAttribute` – Type.

```
AbstractVariableAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of variables in the model.

source

`MathOptInterface.VariableName` – Type.

```
VariableName()
```

A variable attribute for a string identifying the variable. It is valid for two variables to have the same name; however, variables with duplicate names cannot be looked up using `get`. It has a default value of `""` if not set'.

source

`MathOptInterface.VariablePrimalStart` – Type.

```
VariablePrimalStart()
```

A variable attribute for the initial assignment to some primal variable's value that the optimizer may use to warm-start the solve. May be a number or `nothing` (unset).

source

`MathOptInterface.VariablePrimal` – Type.

```
VariablePrimal(result_index::Int = 1)
```

A variable attribute for the assignment to some primal variable's value in result `result_index`. If `result_index` is omitted, it is 1 by default.

If the solver does not have a primal value for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `VariablePrimal` attribute.

See `ResultCount` for information on how the results are ordered.

source

`MathOptInterface.VariableBasisStatus` – Type.

```
VariableBasisStatus(result_index::Int = 1)
```

A variable attribute for the `BasisStatusCode` of a variable in result `result_index`, with respect to an available optimal solution basis.

If the solver does not have a basis statue for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `VariableBasisStatus` attribute.

See `ResultCount` for information on how the results are ordered.

source

Chapter 21

Constraints

21.1 Types

MathOptInterface.ConstraintIndex – Type.

```
ConstraintIndex{F, S}
```

A type-safe wrapper for `Int64` for use in referencing F-in-S constraints in a model. The parameter F is the type of the function in the constraint, and the parameter S is the type of set in the constraint. To allow for deletion, indices need not be consecutive. Indices within a constraint type (i.e. F-in-S) must be unique, but non-unique indices across different constraint types are allowed. If F is `VariableIndex` then the index is equal to the index of the variable. That is for an `index::ConstraintIndex{VariableIndex}`, we always have

```
index.value == MOI.get(model, MOI.ConstraintFunction(), index).value
```

source

21.2 Functions

MathOptInterface.is_valid – Method.

```
is_valid(model::ModelLike, index::Index)::Bool
```

Return a `Bool` indicating whether this index refers to a valid object in the model `model`.

source

MathOptInterface.add_constraint – Function.

```
add_constraint(model::ModelLike, func::F, set::S)::ConstraintIndex{F,S} where {F,S}
```

Add the constraint $f(x) \in \mathcal{S}$ where f is defined by `func`, and $\mathcal{S}$ is defined by `set`.

```
add_constraint(model::ModelLike, v::VariableIndex, set::S)::ConstraintIndex{VariableIndex,S}
    where {S}
add_constraint(model::ModelLike, vec::Vector{VariableIndex}, set::S)::ConstraintIndex{
    VectorOfVariables,S} where {S}
```

Add the constraint $v \in \mathcal{S}$ where v is the variable (or vector of variables) referenced by v and $\mathcal{S}$ is defined by `set`.

- An UnsupportedConstraint error is thrown if model does not support F-in-S constraints,
- a AddConstraintNotAllowed error is thrown if it supports F-in-S constraints but it cannot add the constraint(s) in its current state and
- a ScalarFunctionConstantNotZero error may be thrown if func is an AbstractScalarFunction with nonzero constant and set is EqualTo, GreaterThan, LessThan or Interval.
- a LowerBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets a lower bound.
- a UpperBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets an upper bound.

source

MathOptInterface.add_constraints – Function.

```
add_constraints(model::ModelLike, funcs::Vector{F},
↪  sets::Vector{S})::Vector{ConstraintIndex{F,S}} where {F,S}
```

Add the set of constraints specified by each function-set pair in funcs and sets. F and S should be concrete types. This call is equivalent to add_constraint.(model, funcs, sets) but may be more efficient.

source

MathOptInterface.transform – Function.

Transform Constraint Set

```
transform(model::ModelLike, c::ConstraintIndex{F,S1}, newset::S2)::ConstraintIndex{F,S2}
```

Replace the set in constraint c with newset. The constraint index c will no longer be valid, and the function returns a new constraint index with the correct type.

Solvers may only support a subset of constraint transforms that they perform efficiently (for example, changing from a LessThan to GreaterThan set). In addition, set modification (where S1 = S2) should be performed via the modify function.

Typically, the user should delete the constraint and add a new one.

Examples

If c is a ConstraintIndex{ScalarAffineFunction{Float64},LessThan{Float64}},

```
c2 = transform(model, c, GreaterThan(0.0))
transform(model, c, LessThan(0.0)) # errors
```

source

MathOptInterface.supports_constraint – Function.

```
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool
```

Return a Bool indicating whether the bridges of type BT support bridging F-in-S constraints.

source

```
supports_constraint(
    model::ModelLike,
    ::Type{F},
    ::Type{S},
)::Bool where {F<:AbstractFunction,S<:AbstractSet}
```

Return a `Bool` indicating whether `model` supports F-in-S constraints, that is, `copy_to(model, src)` does not throw `UnsupportedConstraint` when `src` contains F-in-S constraints. If F-in-S constraints are only not supported in specific circumstances, e.g. F-in-S constraints cannot be combined with another type of constraint, it should still return `true`.

source

21.3 Attributes

`MathOptInterface.AbstractConstraintAttribute` – Type.

```
AbstractConstraintAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of constraints in the model.

source

`MathOptInterface.ConstraintName` – Type.

```
ConstraintName()
```

A constraint attribute for a string identifying the constraint.

It is valid for constraints variables to have the same name; however, constraints with duplicate names cannot be looked up using `get`, regardless of whether they have the same F-in-S type.

`ConstraintName` has a default value of `""` if not set.

Notes

You should not implement `ConstraintName` for `VariableIndex` constraints.

source

`MathOptInterface.ConstraintPrimalStart` – Type.

```
ConstraintPrimalStart()
```

A constraint attribute for the initial assignment to some constraint's `ConstraintPrimal` that the optimizer may use to warm-start the solve.

May be `nothing` (unset), a number for `AbstractScalarFunction`, or a vector for `AbstractVectorFunction`.

source

`MathOptInterface.ConstraintDualStart` – Type.

```
ConstraintDualStart()
```

A constraint attribute for the initial assignment to some constraint's `ConstraintDual` that the optimizer may use to warm-start the solve.

May be `nothing` (unset), a number for `AbstractScalarFunction`, or a vector for `AbstractVectorFunction`.

source

MathOptInterface.ConstraintPrimal – Type.

```
ConstraintPrimal(result_index::Int = 1)
```

A constraint attribute for the assignment to some constraint's primal value(s) in result `result_index`.

If the constraint is `f(x) in S`, then in most cases the `ConstraintPrimal` is the value of `f`, evaluated at the correspondng `VariablePrimal` solution.

However, some conic solvers reformulate `b - Ax in S` to `s = b - Ax, s in S`. These solvers may return the value of `s` for `ConstraintPrimal`, rather than `b - Ax`. (Although these are constrained by an equality constraint, due to numerical tolerances they may not be identical.)

If the solver does not have a primal value for the constraint because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `ConstraintPrimal` attribute.

If `result_index` is omitted, it is 1 by default. See `ResultCount` for information on how the results are ordered.

source

MathOptInterface.ConstraintDual – Type.

```
ConstraintDual(result_index::Int = 1)
```

A constraint attribute for the assignment to some constraint's dual value(s) in result `result_index`. If `result_index` is omitted, it is 1 by default.

If the solver does not have a dual value for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a primal solution is available), the result is undefined. Users should first check `DualStatus` before accessing the `ConstraintDual` attribute.

See `ResultCount` for information on how the results are ordered.

source

MathOptInterface.ConstraintBasisStatus – Type.

```
ConstraintBasisStatus(result_index::Int = 1)
```

A constraint attribute for the `BasisStatusCode` of some constraint in result `result_index`, with respect to an available optimal solution basis. If `result_index` is omitted, it is 1 by default.

If the solver does not have a basis statue for the constraint because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `ConstraintBasisStatus` attribute.

See `ResultCount` for information on how the results are ordered.

Notes

For the basis status of a variable, query `VariableBasisStatus`.

ConstraintBasisStatus does not apply to VariableIndex constraints. You can infer the basis status of a VariableIndex constraint by looking at the result of VariableBasisStatus.

source

MathOptInterface.BasisStatusCode – Type.

```
BasisStatusCode
```

An Enum of possible values for the ConstraintBasisStatus and VariableBasisStatus attributes, explaining the status of a given element with respect to an optimal solution basis.

Possible values are:

- BASIC: element is in the basis
- NONBASIC: element is not in the basis
- NONBASIC_AT_LOWER: element is not in the basis and is at its lower bound
- NONBASIC_AT_UPPER: element is not in the basis and is at its upper bound
- SUPER_BASIC: element is not in the basis but is also not at one of its bounds

Notes

- NONBASIC_AT_LOWER and NONBASIC_AT_UPPER should be used only for constraints with the Interval set. In this case, they are necessary to distinguish which side of the constraint is active. One-sided constraints (e.g., LessThan and GreaterThan) should use NONBASIC instead of the NONBASIC_AT_* values. This restriction does not apply to VariableBasisStatus, which should return NONBASIC_AT_* regardless of whether the alternative bound exists.
- In linear programs, SUPER_BASIC occurs when a variable with no bounds is not in the basis.

source

MathOptInterface.ConstraintFunction – Type.

```
ConstraintFunction()
```

A constraint attribute for the AbstractFunction object used to define the constraint. It is guaranteed to be equivalent but not necessarily identical to the function provided by the user.

source

MathOptInterface.CanonicalConstraintFunction – Type.

```
CanonicalConstraintFunction()
```

A constraint attribute for a canonical representation of the AbstractFunction object used to define the constraint. Getting this attribute is guaranteed to return a function that is equivalent but not necessarily identical to the function provided by the user.

By default, MOI.get(model, MOI.CanonicalConstraintFunction(), ci) fallbacks to MOI.Utilities.canonical(MOI.get MOI.ConstraintFunction(), ci)). However, if model knows that the constraint function is canonical then it can implement a specialized method that directly return the function without calling Utilities.canonical. Therefore, the value returned **cannot** be assumed to be a copy of the function stored in model. Moreover, Utilities.Model checks with Utilities.is_canonical whether the function stored internally is already canonical and if it's the case, then it returns the function stored internally instead of a copy.

source

`MathOptInterface.ConstraintSet` – Type.

```
ConstraintSet()
```

A constraint attribute for the `AbstractSet` object used to define the constraint.

source

Chapter 22

Modifications

MathOptInterface.modify – Function.

Constraint Function

```
modify(model::ModelLike, ci::ConstraintIndex, change::AbstractFunctionModification)
```

Apply the modification specified by change to the function of constraint ci.

An ModifyConstraintNotAllowed error is thrown if modifying constraints is not supported by the model model.

Examples

```
modify(model, ci, ScalarConstantChange(10.0))
```

Objective Function

```
modify(model::ModelLike, ::ObjectiveFunction, change::AbstractFunctionModification)
```

Apply the modification specified by change to the objective function of model. To change the function completely, call set instead.

An ModifyObjectiveNotAllowed error is thrown if modifying objectives is not supported by the model model.

Examples

```
modify(model, ObjectiveFunction{ScalarAffineFunction{Float64}}(), ScalarConstantChange(10.0))
```

source

MathOptInterface.AbstractFunctionModification – Type.

```
AbstractFunctionModification
```

An abstract supertype for structs which specify partial modifications to functions, to be used for making small modifications instead of replacing the functions entirely.

source

MathOptInterface.ScalarConstantChange – Type.

```
ScalarConstantChange{T}(new_constant::T)
```

A struct used to request a change in the constant term of a scalar-valued function. Applicable to `ScalarAffineFunction` and `ScalarQuadraticFunction`.

source

`MathOptInterface.VectorConstantChange` – Type.

```
VectorConstantChange{T}(new_constant::Vector{T})
```

A struct used to request a change in the constant vector of a vector-valued function. Applicable to `VectorAffineFunction` and `VectorQuadraticFunction`.

source

`MathOptInterface.ScalarCoefficientChange` – Type.

```
ScalarCoefficientChange{T}(variable::VariableIndex, new_coefficient::T)
```

A struct used to request a change in the linear coefficient of a single variable in a scalar-valued function. Applicable to `ScalarAffineFunction` and `ScalarQuadraticFunction`.

source

`MathOptInterface.MultirowChange` – Type.

```
MultirowChange{T}(variable::VariableIndex, new_coefficients::Vector{Tuple{Int64, T}})
```

A struct used to request a change in the linear coefficients of a single variable in a vector-valued function. New coefficients are specified by `(output_index, coefficient)` tuples. Applicable to `VectorAffineFunction` and `VectorQuadraticFunction`.

source

Chapter 23

Nonlinear programming

23.1 Types

MathOptInterface.AbstractNLPEvaluator – Type.

```
AbstractNLPEvaluator
```

Abstract supertype for the callback object that is used to query function values, derivatives, and expression graphs. It is used in `NLPBlock`.

source

MathOptInterface.NLPBoundsPair – Type.

```
NLPBoundsPair(lower,upper)
```

A struct holding a pair of lower and upper bounds. `-Inf` and `Inf` can be used to indicate no lower or upper bound, respectively.

source

MathOptInterface.NLPBlockData – Type.

```
struct NLPBlockData
    constraint_bounds::Vector{NLPBoundsPair}
    evaluator::AbstractNLPEvaluator
    has_objective::Bool
end
```

A `struct` encoding a set of nonlinear constraints of the form $lb \leq g(x) \leq ub$ and, if `has_objective == true`, a nonlinear objective function $f(x)$. `constraint_bounds` holds the pairs of lb and ub elements. Nonlinear objectives override any objective set by using the `ObjectiveFunction` attribute. The `evaluator` is a callback object that is used to query function values, derivatives, and expression graphs. If `has_objective == false`, then it is an error to query properties of the objective function, and in Hessian-of-the-Lagrangian queries, σ must be set to zero.

> **Note**
>
> Throughout the evaluator, all variables are ordered according to `ListOfVariableIndices`. Hence, MOI copies of nonlinear problems should be done with attention.

source

23.2 Attributes

MathOptInterface.NLPBlock – Type.

```
NLPBlock()
```

Holds the NLPBlockData that represents a set of nonlinear constraints, and optionally a nonlinear objective.

source

MathOptInterface.NLPBlockDual – Type.

```
NLPBlockDual(result_index::Int)
NLPBlockDual()
```

The Lagrange multipliers on the constraints from the NLPBlock in result result_index. If result_index is omitted, it is 1 by default.

source

MathOptInterface.NLPBlockDualStart – Type.

```
NLPBlockDualStart()
```

An initial assignment of the Lagrange multipliers on the constraints from the NLPBlock that the solver may use to warm-start the solve.

source

23.3 Functions

MathOptInterface.initialize – Function.

```
initialize(d::AbstractNLPEvaluator, requested_features::Vector{Symbol})
```

Must be called before any other methods. The vector requested_features lists features requested by the solver. These may include :Grad for gradients of the obejctive, f, :Jac for explicit Jacobians of constraints, g, :JacVec for Jacobian-vector products, :HessVec for Hessian-vector and Hessian-of-Lagrangian-vector products, :Hess for explicit Hessians and Hessian-of-Lagrangians, and :ExprGraph for expression graphs.

source

MathOptInterface.features_available – Function.

```
features_available(d::AbstractNLPEvaluator)
```

Returns the subset of features available for this problem instance, as a vector of symbols in the same format as in initialize.

source

MathOptInterface.eval_objective – Function.

```
eval_objective(d::AbstractNLPEvaluator, x)
```

Evaluate the objective $f(x)$, returning a scalar value.

source

MathOptInterface.eval_constraint – Function.

```
eval_constraint(d::AbstractNLPEvaluator, g, x)
```

Evaluate the constraint function $g(x)$, storing the result in the vector g which must be of the appropriate size.

source

MathOptInterface.eval_objective_gradient – Function.

```
eval_objective_gradient(d::AbstractNLPEvaluator, df, x)
```

Evaluate $\nabla f(x)$ as a dense vector, storing the result in the vector `df` which must be of the appropriate size.

source

MathOptInterface.jacobian_structure – Function.

```
jacobian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}
```

Returns the sparsity structure of the Jacobian matrix $J_g(x) = \left[\begin{array}{c} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{array}\right]$ where g_i is the ith component of g. The sparsity structure is assumed to be independent of the point x. Returns a vector of tuples, `(row, column)`, where each indicates the position of a structurally nonzero element. These indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

source

MathOptInterface.hessian_lagrangian_structure – Function.

```
hessian_lagrangian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}
```

Returns the sparsity structure of the Hessian-of-the-Lagrangian matrix $\nabla^2 f + \sum_{i=1}^m \nabla^2 g_i$ as a vector of tuples, where each indicates the position of a structurally nonzero element. These indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together. Any mix of lower and upper-triangular indices is valid. Elements `(i,j)` and `(j,i)`, if both present, should be treated as duplicates.

source

MathOptInterface.eval_constraint_jacobian – Function.

```
eval_constraint_jacobian(d::AbstractNLPEvaluator, J, x)
```

Evaluates the sparse Jacobian matrix $J_g(x) = \left[\begin{array}{c} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{array}\right]$. The result is stored in the vector J in the same order as the indices returned by `jacobian_structure`.

source

`MathOptInterface.eval_constraint_jacobian_product` – Function.

```
eval_constraint_jacobian_product(d::AbstractNLPEvaluator, y, x, w)
```

Computes the Jacobian-vector product $J_g(x)w$, storing the result in the vector `y`.

source

`MathOptInterface.eval_constraint_jacobian_transpose_product` – Function.

```
eval_constraint_jacobian_transpose_product(d::AbstractNLPEvaluator, y, x, w)
```

Computes the Jacobian-transpose-vector product $J_g(x)^T w$, storing the result in the vector `y`.

source

`MathOptInterface.eval_hessian_lagrangian` – Function.

```
eval_hessian_lagrangian(d::AbstractNLPEvaluator, H, x, σ, μ)
```

Given scalar weight `σ` and vector of constraint weights `μ`, computes the sparse Hessian-of-the-Lagrangian matrix $\sigma\nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)$, storing the result in the vector `H` in the same order as the indices returned by `hessian_lagrangian_structure`.

source

`MathOptInterface.eval_hessian_lagrangian_product` – Function.

```
eval_hessian_lagrangian_product(d::AbstractNLPEvaluator, h, x, v, σ, μ)
```

Given scalar weight `σ` and vector of constraint weights `μ`, computes the Hessian-of-the-Lagrangian-vector product $\left(\sigma\nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)\right) v$, storing the result in the vector `h`.

source

`MathOptInterface.objective_expr` – Function.

```
objective_expr(d::AbstractNLPEvaluator)
```

Returns an expression graph for the objective function as a standard Julia `Expr` object. All sums and products are flattened out as simple `Expr(:+,...)` and `Expr(:*,...)` objects. The symbol `x` is used as a placeholder for the vector of decision variables. No other undefined symbols are permitted; coefficients are embedded as explicit values. For example, the expression $x_1 + \sin(x_2/\exp(x_3))$ would be represented as the Julia object `:(x[1] + sin(x[2]/exp(x[3])))`. Each integer index is wrapped in a `VariableIndex`. See the Julia manual for more information on the structure of `Expr` objects. There are currently no restrictions on recognized functions; typically these will be built-in Julia functions like `^`, `exp`, `log`, `cos`, `tan`, `sqrt`, etc., but modeling interfaces may choose to extend these basic functions.

source

`MathOptInterface.constraint_expr` – Function.

```
constraint_expr(d::AbstractNLPEvaluator, i)
```

Returns an expression graph for the ith constraint in the same format as described above, with an additional comparison operator indicating the sense of and bounds on the constraint. The right-hand side of the comparison must be a constant; that is, `:(x[1]^3 <= 1)` is allowed, while `:(1 <= x[1]^3)` is not valid. Double-sided constraints are allowed, in which case both the lower bound and upper bounds should be constants; for example, `:(-1 <= cos(x[1]) + sin(x[2]) <= 1)` is valid.

source

Chapter 24

Callbacks

MathOptInterface.AbstractCallback – Type.

```
abstract type AbstractCallback <: AbstractModelAttribute end
```

Abstract type for a model attribute representing a callback function. The value set to subtypes of AbstractCallback is a function that may be called during optimize!. As optimize! is in progress, the result attributes (i.e, the attributes attr such that is_set_by_optimize(attr)) may not be accessible from the callback, hence trying to get result attributes might throw a OptimizeInProgress error.

At most one callback of each type can be registered. If an optimizer already has a function for a callback type, and the user registers a new function, then the old one is replaced.

The value of the attribute should be a function taking only one argument, commonly called callback_data, that can be used for instance in LazyConstraintCallback, HeuristicCallback and UserCutCallback.

source

MathOptInterface.AbstractSubmittable – Type.

```
AbstractSubmittable
```

Abstract supertype for objects that can be submitted to the model.

source

MathOptInterface.submit – Function.

```
submit(optimizer::AbstractOptimizer, sub::AbstractSubmittable,
       values...)::Nothing
```

Submit values to the submittable sub of the optimizer optimizer.

An UnsupportedSubmittable error is thrown if model does not support the attribute attr (see supports) and a SubmitNotAllowed error is thrown if it supports the submittable sub but it cannot be submitted.

source

24.1 Attributes

MathOptInterface.CallbackNodeStatus – Type.

```
CallbackNodeStatus(callback_data)
```

An optimizer attribute describing the (in)feasibility of the primal solution available from CallbackVariablePrimal during a callback identified by callback_data.

Returns a CallbackNodeStatusCode Enum.

source

MathOptInterface.CallbackNodeStatusCode – Type.

```
CallbackNodeStatusCode
```

An Enum of possible return values from calling get with CallbackNodeStatus.

Possible values are:

- CALLBACK_NODE_STATUS_INTEGER: the primal solution available from CallbackVariablePrimal is integer feasible.
- CALLBACK_NODE_STATUS_FRACTIONAL: the primal solution available from CallbackVariablePrimal is integer infeasible.
- CALLBACK_NODE_STATUS_UNKNOWN: the primal solution available from CallbackVariablePrimal might be integer feasible or infeasible.

source

MathOptInterface.CallbackVariablePrimal – Type.

```
CallbackVariablePrimal(callback_data)
```

A variable attribute for the assignment to some primal variable's value during the callback identified by callback_data.

source

24.2 Lazy constraints

MathOptInterface.LazyConstraintCallback – Type.

```
LazyConstraintCallback() <: AbstractCallback
```

The callback can be used to reduce the feasible set given the current primal solution by submitting a LazyConstraint. For instance, it may be called at an incumbent of a mixed-integer problem. Note that there is no guarantee that the callback is called at every feasible primal solution.

The current primal solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.LazyConstraintCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # should add a lazy constraint
        func = # computes function
        set = # computes set
        MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
    end
end)
```

source

MathOptInterface.LazyConstraint – Type.

```
LazyConstraint(callback_data)
```

Lazy constraint func-in-set submitted as func, set. The optimal solution returned by VariablePrimal will satisfy all lazy constraints that have been submitted.

This can be submitted only from the LazyConstraintCallback. The field callback_data is a solver-specific callback type that is passed as the argument to the feasible solution callback.

Examples

Suppose x and y are VariableIndexs of optimizer. To add a LazyConstraint for 2x + 3y <= 1, write

```
func = 2.0x + 3.0y
set = MOI.LessThan(1.0)
MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
```

inside a LazyConstraintCallback of data callback_data.

source

24.3 User cuts

MathOptInterface.UserCutCallback – Type.

```
UserCutCallback() <: AbstractCallback
```

The callback can be used to submit UserCut given the current primal solution. For instance, it may be called at fractional (i.e., non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is not guarantee that the callback is called everytime the solver has an infeasible solution.

The infeasible solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.UserCutCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # can find a user cut
        func = # computes function
        set = # computes set
        MOI.submit(optimizer, MOI.UserCut(callback_data), func, set)
    end
end
```

source

MathOptInterface.UserCut – Type.

```
UserCut(callback_data)
```

Constraint `func-to-set` suggested to help the solver detect the solution given by `CallbackVariablePrimal` as infeasible. The cut is submitted as `func, set`. Typically `CallbackVariablePrimal` will violate integrality constraints, and a cut would be of the form `ScalarAffineFunction`-in-`LessThan` or `ScalarAffineFunction`-in-`GreaterThan`. Note that, as opposed to `LazyConstraint`, the provided constraint cannot modify the feasible set, the constraint should be redundant, e.g., it may be a consequence of affine and integrality constraints.

This can be submitted only from the `UserCutCallback`. The field `callback_data` is a solver-specific callback type that is passed as the argument to the infeasible solution callback.

Note that the solver may silently ignore the provided constraint.

source

24.4 Heuristic solutions

`MathOptInterface.HeuristicCallback` – Type.

```
HeuristicCallback() <: AbstractCallback
```

The callback can be used to submit `HeuristicSolution` given the current primal solution. For instance, it may be called at fractional (i.e., non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is not guarantee that the callback is called everytime the solver has an infeasible solution.

The current primal solution is accessed through `CallbackVariablePrimal`. Trying to access other result attributes will throw `OptimizeInProgress` as discussed in `AbstractCallback`.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.HeuristicCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # can find a heuristic solution
        values = # computes heuristic solution
        MOI.submit(optimizer, MOI.HeuristicSolution(callback_data), x,
                   values)
    end
end
```

source

`MathOptInterface.HeuristicSolutionStatus` – Type.

```
HeuristicSolutionStatus
```

An Enum of possible return values for `submit` with `HeuristicSolution`. This informs whether the heuristic solution was accepted or rejected. Possible values are:

- `HEURISTIC_SOLUTION_ACCEPTED`: The heuristic solution was accepted.
- `HEURISTIC_SOLUTION_REJECTED`: The heuristic solution was rejected.
- `HEURISTIC_SOLUTION_UNKNOWN`: No information available on the acceptance.

source

`MathOptInterface.HeuristicSolution` – Type.

```
HeuristicSolution(callback_data)
```

Heuristically obtained feasible solution. The solution is submitted as `variables, values` where `values[i]` gives the value of `variables[i]`, similarly to `set`. The `submit` call returns a `HeuristicSolutionStatus` indicating whether the provided solution was accepted or rejected.

This can be submitted only from the `HeuristicCallback`. The field `callback_data` is a solver-specific callback type that is passed as the argument to the heuristic callback.

Some solvers require a complete solution, others only partial solutions.

source

Chapter 25

Errors

When an MOI call fails on a model, precise errors should be thrown when possible instead of simply calling `error` with a message. The docstrings for the respective methods describe the errors that the implementation should throw in certain situations. This error-reporting system allows code to distinguish between internal errors (that should be shown to the user) and unsupported operations which may have automatic workarounds.

When an invalid index is used in an MOI call, an `InvalidIndex` is thrown:

`MathOptInterface.InvalidIndex` – Type.

```
struct InvalidIndex{IndexType<:Index} <: Exception
    index::IndexType
end
```

An error indicating that the index `index` is invalid.

source

When an invalid result index is used to retrieve an attribute, a `ResultIndexBoundsError` is thrown:

`MathOptInterface.ResultIndexBoundsError` – Type.

```
struct ResultIndexBoundsError{AttrType} <: Exception
    attr::AttrType
    result_count::Int
end
```

An error indicating that the requested attribute `attr` could not be retrieved, because the solver returned too few results compared to what was requested. For instance, the user tries to retrieve `VariablePrimal(2)` when only one solution is available, or when the model is infeasible and has no solution.

See also: `check_result_index_bounds`.

source

`MathOptInterface.check_result_index_bounds` – Function.

```
check_result_index_bounds(model::ModelLike, attr)
```

This function checks whether enough results are available in the `model` for the requested `attr`, using its `result_index` field. If the model does not have sufficient results to answer the query, it throws a `ResultIndexBoundsError`.

source

As discussed in JuMP mapping, for scalar constraint with a nonzero function constant, a ScalarFunctionConstantNotZero exception may be thrown:

MathOptInterface.ScalarFunctionConstantNotZero – Type.

```
struct ScalarFunctionConstantNotZero{T, F, S} <: Exception
    constant::T
end
```

An error indicating that the constant part of the function in the constraint F-in-S is nonzero.

source

Some VariableIndex constraints cannot be combined on the same variable:

MathOptInterface.LowerBoundAlreadySet – Type.

```
LowerBoundAlreadySet{S1, S2}
```

Error thrown when setting a VariableIndex-in-S2 when a VariableIndex-in-S1 has already been added and the sets S1, S2 both set a lower bound, i.e. they are EqualTo, GreaterThan, Interval, Semicontinuous or Semiinteger.

source

MathOptInterface.UpperBoundAlreadySet – Type.

```
UpperBoundAlreadySet{S1, S2}
```

Error thrown when setting a VariableIndex-in-S2 when a VariableIndex-in-S1 has already been added and the sets S1, S2 both set an upper bound, i.e. they are EqualTo, LessThan, Interval, Semicontinuous or Semiinteger.

source

As discussed in AbstractCallback, trying to get attributes inside a callback may throw:

MathOptInterface.OptimizeInProgress – Type.

```
struct OptimizeInProgress{AttrType<:AnyAttribute} <: Exception
    attr::AttrType
end
```

Error thrown from optimizer when `MOI.get(optimizer, attr)` is called inside an AbstractCallback while it is only defined once optimize! has completed. This can only happen when `is_set_by_optimize(attr)` is `true`.

source

Trying to submit the wrong type of AbstractSubmittable inside an AbstractCallback (e.g., a UserCut inside a LazyConstraintCallback) will throw:

MathOptInterface.InvalidCallbackUsage – Type.

```
struct InvalidCallbackUsage{C, S} <: Exception
    callback::C
    submittable::S
end
```

An error indicating that `submittable` cannot be submitted inside `callback`.

For example, `UserCut` cannot be submitted inside `LazyConstraintCallback`.

source

The rest of the errors defined in MOI fall in two categories represented by the following two abstract types:

`MathOptInterface.UnsupportedError` – Type.

```
UnsupportedError <: Exception
```

Abstract type for error thrown when an element is not supported by the model.

source

`MathOptInterface.NotAllowedError` – Type.

```
NotAllowedError <: Exception
```

Abstract type for error thrown when an operation is supported but cannot be applied in the current state of the model.

source

The different `UnsupportedError` and `NotAllowedError` are the following errors:

`MathOptInterface.UnsupportedAttribute` – Type.

```
struct UnsupportedAttribute{AttrType} <: UnsupportedError
    attr::AttrType
    message::String
end
```

An error indicating that the attribute `attr` is not supported by the model, i.e. that `supports` returns `false`.

source

`MathOptInterface.SetAttributeNotAllowed` – Type.

```
struct SetAttributeNotAllowed{AttrType} <: NotAllowedError
    attr::AttrType
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that the attribute `attr` is supported (see `supports`) but cannot be set for some reason (see the error string).

source

`MathOptInterface.AddVariableNotAllowed` – Type.

```
struct AddVariableNotAllowed <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that variables cannot be added to the model.

source

`MathOptInterface.UnsupportedConstraint` – Type.

```
struct UnsupportedConstraint{F<:AbstractFunction, S<:AbstractSet} <: UnsupportedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that constraints of type F-in-S are not supported by the model, i.e. that `supports_constraint` returns `false`.

source

`MathOptInterface.AddConstraintNotAllowed` – Type.

```
struct AddConstraintNotAllowed{F<:AbstractFunction, S<:AbstractSet} <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that constraints of type F-in-S are supported (see `supports_constraint`) but cannot be added.

source

`MathOptInterface.ModifyConstraintNotAllowed` – Type.

```
struct ModifyConstraintNotAllowed{F<:AbstractFunction, S<:AbstractSet,
                                  C<:AbstractFunctionModification} <: NotAllowedError
    constraint_index::ConstraintIndex{F, S}
    change::C
    message::String
end
```

An error indicating that the constraint modification `change` cannot be applied to the constraint of index `ci`.

source

`MathOptInterface.ModifyObjectiveNotAllowed` – Type.

```
struct ModifyObjectiveNotAllowed{C<:AbstractFunctionModification} <: NotAllowedError
    change::C
    message::String
end
```

An error indicating that the objective modification `change` cannot be applied to the objective.

source

`MathOptInterface.DeleteNotAllowed` – Type.

```
struct DeleteNotAllowed{IndexType <: Index} <: NotAllowedError
    index::IndexType
    message::String
end
```

An error indicating that the index `index` cannot be deleted.

source

`MathOptInterface.UnsupportedSubmittable` – Type.

```
struct UnsupportedSubmittable{SubmitType} <: UnsupportedError
    sub::SubmitType
    message::String
end
```

An error indicating that the submittable `sub` is not supported by the model, i.e. that `supports` returns `false`.

source

`MathOptInterface.SubmitNotAllowed` – Type.

```
struct SubmitNotAllowed{SubmitTyp<:AbstractSubmittable} <: NotAllowedError
    sub::SubmitType
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that the submittable `sub` is supported (see `supports`) but cannot be added for some reason (see the error string).

source

Note that setting the `ConstraintFunction` of a `VariableIndex` constraint is not allowed:

`MathOptInterface.SettingVariableIndexNotAllowed` – Type.

```
SettingVariableIndexNotAllowed()
```

Error type that should be thrown when the user calls `set` to change the `ConstraintFunction` of a `VariableIndex` constraint.

source

Part VI

Submodules

Chapter 26

Benchmarks

26.1 Overview

The `Benchmarks` submodule

To aid the development of efficient solver wrappers, MathOptInterface provides benchmarking functionality. Benchmarking a wrapper follows a two-step process.

First, prior to making changes, run and save the benchmark results on a given benchmark suite as follows:

```julia
using SolverPackage  # Replace with your choice of solver.

using MathOptInterface
const MOI = MathOptInterface

suite = MOI.Benchmarks.suite() do
    SolverPackage.Optimizer()
end

MOI.Benchmarks.create_baseline(
    suite, "current"; directory = "/tmp", verbose = true
)
```

Use the `exclude` argument to `Benchmarks.suite` to exclude benchmarks that the solver doesn't support.

Second, after making changes to the package, re-run the benchmark suite and compare to the prior saved results:

```julia
using SolverPackage, MathOptInterface

const MOI = MathOptInterface

suite = MOI.Benchmarks.suite() do
    SolverPackage.Optimizer()
end

MOI.Benchmarks.compare_against_baseline(
    suite, "current"; directory = "/tmp", verbose = true
)
```

This comparison will create a report detailing improvements and regressions.

26.2 API Reference

Benchmarks

Functions to help benchmark the performance of solver wrappers. See The Benchmarks submodule for more details.

MathOptInterface.Benchmarks.suite – Function.

```
suite(
    new_model::Function;
    exclude::Vector{Regex} = Regex[]
)
```

Create a suite of benchmarks. new_model should be a function that takes no arguments, and returns a new instance of the optimizer you wish to benchmark.

Use exclude to exclude a subset of benchmarks.

Examples

```
suite() do
    GLPK.Optimizer()
end
suite(exclude = [r"delete"]) do
    Gurobi.Optimizer(OutputFlag=0)
end
```

source

MathOptInterface.Benchmarks.create_baseline – Function.

```
create_baseline(suite, name::String; directory::String = ""; kwargs...)
```

Run all benchmarks in suite and save to files called name in directory.

Extra kwargs are based to BenchmarkTools.run.

Examples

```
my_suite = suite(() -> GLPK.Optimizer())
create_baseline(my_suite, "glpk_master"; directory = "/tmp", verbose = true)
```

source

MathOptInterface.Benchmarks.compare_against_baseline – Function.

```
compare_against_baseline(
    suite, name::String; directory::String = "",
    report_filename::String = "report.txt"
)
```

Run all benchmarks in suite and compare against files called name in directory that were created by a call to create_baseline.

A report summarizing the comparison is written to report_filename in directory.

Extra kwargs are based to BenchmarkTools.run.

Examples

```
my_suite = suite(() -> GLPK.Optimizer())
compare_against_baseline(
    my_suite, "glpk_master"; directory = "/tmp", verbose = true
)
```

source

Chapter 27

Bridges

27.1 Overview

The Bridges submodule

The `Bridges` module simplifies the process of converting models between equivalent formulations.

> **Tip**
>
> Read our paper for more details on how bridges are implemented.

Why bridges?

A constraint can often be written in a number of equivalent formulations. For example, the constraint $l \leq a^\top x \leq u$ (`ScalarAffineFunction`-in-`Interval`) could be re-formulated as two constraints: $a^\top x \geq l$ (`ScalarAffineFunction`-in-`GreaterThan`) and $a^\top x \leq u$ (`ScalarAffineFunction`-in-`LessThan`). An alternative re-formulation is to add a dummy variable `y` with the constraints $l \leq y \leq u$ (`VariableIndex`-in-`Interval`) and $a^\top x - y = 0$ (`ScalarAffineFunction`-in-`EqualTo`).

To avoid each solver having to code these transformations manually, MathOptInterface provides bridges.

A bridge is a small transformation from one constraint type to another (potentially collection of) constraint type.

Because these bridges are included in MathOptInterface, they can be re-used by any optimizer. Some bridges also implement constraint modifications and constraint primal and dual translations.

Several bridges can be used in combination to transform a single constraint into a form that the solver may understand. Choosing the bridges to use takes the form of finding a shortest path in the hypergraph of bridges. The methodology is detailed in the MOI paper.

The three types of bridges

There are three types of bridges in MathOptInterface:

1. Constraint bridges
2. Variable bridges
3. Objective bridges

Constraint bridges Constraint bridges convert constraints formulated by the user into an equivalent form supported by the solver. Constraint bridges are subtypes of `Bridges.Constraint.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

In particular, constraint bridges can focus on rewriting the function of a constraint, and do not change the set. Function bridges are subtypes of `Bridges.Constraint.AbstractFunctionConversionBridge`.

Read the list of implemented constraint bridges for more details on the types of transformations that are available. Function bridges are `Bridges.Constraint.ScalarFunctionizeBridge` and `Bridges.Constraint.VectorFunctionizeBri`

Variable bridges Variable bridges convert variables added by the user, either free with `add_variable`/`add_variables`, or constrained with `add_constrained_variable`/`add_constrained_variables`, into an equivalent form supported by the solver. Variable bridges are subtypes of `Bridges.Variable.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented variable bridges for more details on the types of transformations that are available.

Objective bridges Objective bridges convert the `ObjectiveFunction` set by the user into an equivalent form supported by the solver. Objective bridges are subtypes of `Bridges.Objective.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented objective bridges for more details on the types of transformations that are available.

Bridges.`full_bridge_optimizer`

> **Tip**
>
> Unless you have an advanced use-case, this is probably the only function you need to care about.

To enable the full power of MathOptInterface's bridges, wrap an `optimizer` in a `Bridges.full_bridge_optimizer`.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.full_bridge_optimizer(inner_optimizer, Float64)
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
with 0 variable bridges
with 0 constraint bridges
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

That's all you have to do! Use `optimizer` as normal, and bridging will happen lazily behind the scenes. By lazily, we mean that bridging will only happen if the constraint is not supported by the `inner_optimizer`.

> **Info**
>
> Most bridges are added by default in `Bridges.full_bridge_optimizer`. However, for technical reasons, some bridges are not added by default. Three examples include `Bridges.Constraint.SOCtoPSDBridge`, `Bridges.Constraint.SOCtoNonConvexQuadBridge` and `Bridges.Constraint.RSOCtoNonConvexQuadBridge`. See the docs of those bridges for more information.

Add a single bridge

If you don't want to use `Bridges.full_bridge_optimizer`, you can wrap an optimizer in a single bridge.

However, this will force the constraint to be bridged, even if the `inner_optimizer` supports it.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.Constraint.SplitInterval{Float64}(inner_optimizer)
MOIB.Constraint.SingleBridgeOptimizer{MOIB.Constraint.SplitIntervalBridge{Float64, F, S, LS, US}
↪  where {F<:MOI.AbstractFunction, S<:MOI.AbstractSet, LS<:MOI.AbstractSet, US<:MOI.AbstractSet},
↪  MOIU.Model{Float64}}
with 0 constraint bridges
with inner model MOIU.Model{Float64}

julia> x = MOI.add_variable(optimizer)
MOI.VariableIndex(1)

julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}}(1)

julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})

julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent())
2-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.GreaterThan{Float64})
 (MathOptInterface.VariableIndex, MathOptInterface.LessThan{Float64})
```

Bridges.LazyBridgeOptimizer

If you don't want to use `Bridges.full_bridge_optimizer`, but you need more than a single bridge (or you want the bridging to happen lazily), you can manually construct a `Bridges.LazyBridgeOptimizer`.

First, wrap an inner optimizer:

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.LazyBridgeOptimizer(inner_optimizer)
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
with 0 variable bridges
with 0 constraint bridges
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

Then use `Bridges.add_bridge` to add individual bridges:

```
julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Constraint.SplitIntervalBridge{Float64})

julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Objective.FunctionizeBridge{Float64})
```

Now the constraints will be bridged only if needed:

```
julia> x = MOI.add_variable(optimizer)
MOI.VariableIndex(1)

julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}}(1)

julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})

julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})
```

27.2 Implementation

Bridge interface

To be usable by a bridge optimizer, a bridge must implement the following functions:

`MathOptInterface.Bridges.added_constrained_variable_types` – Function.

```
added_constrained_variable_types(
    BT::Type{<:Variable.AbstractBridge},
)::Vector{Tuple{Type}}
```

Return a list of the types of constrained variables that bridges of concrete type BT add. This is used by the `LazyBridgeOptimizer`.

source

`MathOptInterface.Bridges.added_constraint_types` – Function.

```
added_constraint_types(
    BT::Type{<:Constraint.AbstractBridge},
)::Vector{Tuple{Type, Type}}
```

Return a list of the types of constraints that bridges of concrete type BT add. This is used by the `LazyBridgeOptimizer`.

source

Additionally, variable bridges must implement:

`MathOptInterface.Bridges.Variable.supports_constrained_variable` – Function.

```
supports_constrained_variable(
    ::Type{<:AbstractBridge},
    ::Type{<:MOI.AbstractSet},
)::Bool
```

Return a `Bool` indicating whether the bridges of type BT support bridging constrained variables in S.

source

`MathOptInterface.Bridges.Variable.concrete_bridge_type` – Function.

```julia
concrete_bridge_type(
    BT::Type{<:AbstractBridge},
    S::Type{<:MOI.AbstractSet},
)::Type
```

Return the concrete type of the bridge supporting variables in S constraints. This function can only be called if `MOI.supports_constrained_variable(BT, S)` is `true`.

Examples

As a variable in `MathOptInterface.GreaterThan` is bridged into variables in `MathOptInterface.Nonnegatives` by the `VectorizeBridge`:

```julia
MOI.Bridges.Variable.concrete_bridge_type(
    MOI.Bridges.Variable.VectorizeBridge{Float64},
    MOI.GreaterThan{Float64},
)

# output

MathOptInterface.Bridges.Variable.VectorizeBridge{Float64, MathOptInterface.Nonnegatives}
```

source

`MathOptInterface.Bridges.Variable.bridge_constrained_variable` – Function.

```julia
bridge_constrained_variable(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    set::MOI.AbstractSet,
)
```

Bridge the constrained variable in `set` using bridge BT to `model` and returns a bridge object of type BT. The bridge type BT should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for given set types.

source

constraint bridges must implement:

`MathOptInterface.supports_constraint` – Method.

```julia
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool
```

Return a `Bool` indicating whether the bridges of type BT support bridging F-in-S constraints.

source

`MathOptInterface.Bridges.Constraint.concrete_bridge_type` – Function.

```julia
concrete_bridge_type(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet}
)::Type
```

Return the concrete type of the bridge supporting F-in-S constraints. This function can only be called if `MOI.supports_constraint(BT, F, S)` is `true`.

Examples

As a `MathOptInterface.VariableIndex`-in-`MathOptInterface.Interval` constraint is bridged into a `MathOptInterface.Var`
in-`MathOptInterface.GreaterThan` and a `MathOptInterface.VariableIndex`-in-`MathOptInterface.LessThan` by the `SplitIntervalBridge`:

```
MOI.Bridges.Constraint.concrete_bridge_type(
    MOI.Bridges.Constraint.SplitIntervalBridge{Float64},
    MOI.VariableIndex,
    MOI.Interval{Float64},
)

# output

MathOptInterface.Bridges.Constraint.SplitIntervalBridge{Float64, MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}, MathOptInterface.GreaterThan{Float64},
↪  MathOptInterface.LessThan{Float64}}
```

source

`MathOptInterface.Bridges.Constraint.bridge_constraint` – Function.

```
bridge_constraint(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    func::AbstractFunction,
    set::MOI.AbstractSet,
)
```

Bridge the constraint `func`-in-`set` using bridge BT to `model` and returns a bridge object of type BT. The bridge type BT should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for given function and set types.

source

and objective bridges must implement:

`MathOptInterface.Bridges.set_objective_function_type` – Function.

```
set_objective_function_type(
    BT::Type{<:Objective.AbstractBridge},
)::Type{<:MOI.AbstractScalarFunction}
```

Return the type of objective function that bridges of concrete type BT set. This is used by the `LazyBridgeOptimizer`.

source

`MathOptInterface.Bridges.Objective.concrete_bridge_type` – Function.

```
concrete_bridge_type(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractScalarFunction},
)::Type
```

Return the concrete type of the bridge supporting objective functions of type F. This function can only be called if `MOI.supports_objective_function(BT, F)` is `true`.

source

MathOptInterface.Bridges.Objective.bridge_objective – Function.

```
bridge_objective(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    model::MOI.ModelLike,
    func::MOI.AbstractScalarFunction,
)
```

Bridge the objective function `func` using bridge `BT` to `model` and returns a bridge object of type `BT`. The bridge type `BT` should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for a given function type.

source

When querying the `NumberOfVariables`, `NumberOfConstraints` `ListOfVariableIndices`, and `ListOfConstraintIndices`, the variables and constraints created by the bridges in the underlying model are hidden by the bridge optimizer. For this purpose, the bridge must provide access to the variables and constraints it has created by implementing the following methods of `get`:

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)
```

The number of variables created by the bridge `b` in the model.

source

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)
```

The list of variables created by the bridge `b` in the model.

source

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F, S}) where {F, S}
```

The number of constraints of the type F-in-S created by the bridge `b` in the model.

source

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F, S}) where {F, S}
```

A `Vector{ConstraintIndex{F,S}}` with indices of all constraints of type F-inS created by the bride `b` in the model (i.e., of length equal to the value of `NumberOfConstraints{F,S}()`).

source

SetMap bridges

Implementing a constraint bridge relying on linear transformation between two sets is easier thanks to the SetMap interface. The bridge simply needs to be a subtype of [`Bridges.Variable.SetMapBridge`] for a variable bridge and [`Bridges.Constraint.SetMapBridge`] for a constraint bridge and the linear transformation is represented with `Bridges.map_set`, `Bridges.map_function`, `Bridges.inverse_map_set`, `Bridges.inverse_map_function`, Bridges.adjoint_map_function and Bridges.inverse_adjoint_map_function. Note that the implementing last 4 methods is optional in the sense that if they are not implemented, bridging constraint would still work but some features would be missing as described in the docstrings. See [L20, Section 2.1.2] for more details including [L20, Example 2.1.1] that illustrates the idea for Bridges.Variable.SOCtoRSOCBridge, Bridges.Variable.RSOCtoSOCBr Bridges.Constraint.SOCtoRSOCBridge and Bridges.Constraint.RSOCtoSOCBridge.

[L20] Legat, Benoît. Set Programming: Theory and Computation. PhD thesis. 2020.

27.3 API Reference

Bridges

MathOptInterface.Bridges.AbstractBridge – Type.

```
AbstractBridge
```

Represents a bridged constraint or variable in a MathOptInterface.Bridges.AbstractBridgeOptimizer. It contains the indices of the variables and constraints that it has created in the model. These can be obtained using MathOptInterface.NumberOfVariables, MathOptInterface.ListOfVariableIndices, MathOptInterface.Nu and MathOptInterface.ListOfConstraintIndices using MathOptInterface.get with the bridge in place of the MathOptInterface.ModelLike. Attributes of the bridged model such as MathOptInterface.ConstraintDual and MathOptInterface.ConstraintPrimal, can be obtained using MathOptInterface.get with the bridge in place of the constraint index. These calls are used by the MathOptInterface.Bridges.AbstractBridgeOptimizer to communicate with the bridge so they should be implemented by the bridge.

source

MathOptInterface.Bridges.AbstractBridgeOptimizer – Type.

```
AbstractBridgeOptimizer
```

A bridge optimizer applies given constraint bridges to a given optimizer thus extending the types of supported constraints. The attributes of the inner optimizer are automatically transformed to make the bridges transparent, e.g. the variables and constraints created by the bridges are hidden.

By convention, the inner optimizer should be stored in a `model` field and the dictionary mapping constraint indices to bridges should be stored in a `bridges` field. If a bridge optimizer deviates from these conventions, it should implement the functions `MOI.optimize!` and `bridge` respectively.

source

MathOptInterface.Bridges.LazyBridgeOptimizer – Type.

```
LazyBridgeOptimizer{OT<:MOI.ModelLike} <: AbstractBridgeOptimizer
```

The `LazyBridgeOptimizer` combines several bridges, which are added using the add_bridge function.

Whenever a constraint is added, it only attempts to bridge it if it is not supported by the internal model (hence its name Lazy).

When bridging a constraint, it selects the minimal number of bridges needed.

For example, if a constraint F-in-S can be bridged into a constraint F1-in-S1 (supported by the internal model) using bridge 1 or bridged into a constraint F2-in-S2 (unsupported by the internal model) using bridge 2 which can then be bridged into a constraint F3-in-S3 (supported by the internal model) using bridge 3, it will choose bridge 1 as it allows to bridge F-in-‘S using only one bridge instead of two if it uses bridge 2 and 3.

source

MathOptInterface.Bridges.add_bridge – Function.

```
add_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Enable the use of the bridges of type BT by b.

source

MathOptInterface.Bridges.remove_bridge – Function.

```
remove_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Disable the use of the bridges of type BT by b.

source

MathOptInterface.Bridges.has_bridge – Function.

```
has_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Return a `Bool` indicating whether the bridges of type BT are used by b.

source

MathOptInterface.Bridges.full_bridge_optimizer – Function.

```
full_bridge_optimizer(model::MOI.ModelLike, ::Type{T}) where {T}
```

Returns a `LazyBridgeOptimizer` bridging `model` for every bridge defined in this package (see below for the few exceptions) and for the coefficient type T in addition to the bridges in the list returned by `MOI.get(model, MOI.Bridges.ListOfNonstandardBridges{T}())`.

See also `ListOfNonstandardBridges`.

Note

The following bridges are not added by `full_bridge_optimizer` except if they are in the list returned by `MOI.get(model, MOI.Bridges.ListOfNonstandardBridges{T}())` (see the docstrings of the corresponding bridge for the reason they are not added):

- `Constraint.SOCtoNonConvexQuadBridge`, `Constraint.RSOCtoNonConvexQuadBridge` and `Constraint.SOCtoPSDBridge`.
- The subtypes of `Constraint.AbstractToIntervalBridge` (i.e. `Constraint.GreaterToIntervalBridge` and `Constraint.LessToIntervalBridge`) if T is not a subtype of `AbstractFloat`.

source

MathOptInterface.Bridges.ListOfNonstandardBridges – Type.

```
ListOfNonstandardBridges{T}() <: MOI.AbstractOptimizerAttribute
```

Any optimizer can be wrapped in a `LazyBridgeOptimizer` using `full_bridge_optimizer`. However, by default `LazyBridgeOptimizer` uses a limited set of bridges that are:

1. implemented in `MOI.Bridges`
2. generally applicable for all optimizers.

For some optimizers however, it is useful to add additional bridges, such as those that are implemented in external packages (e.g., within the solver package itself) or only apply in certain circumstances (e.g., `Constraint.SOCtoNonConvexQuadBridge`).

Such optimizers should implement the `ListOfNonstandardBridges` attribute to return a vector of bridge types that are added by `full_bridge_optimizer` in addition to the list of default bridges.

Note that optimizers implementing `ListOfNonstandardBridges` may require package-specific functions or sets to be used if the non-standard bridges are not added. Therefore, you are recommended to use `model = MOI.instantiate(Package.Optimizer; with_bridge_type = T)` instead of `model = MOI.instantiate(Package.Optimiz`
See `MathOptInterface.instantiate`.

Examples

An optimizer using a non-default bridge in `MOI.Bridges`

Solvers supporting `MOI.ScalarQuadraticFunction` can support `MOI.SecondOrderCone` and `MOI.RotatedSecondOrderCone` by defining:

```
function MOI.get(::MyQuadraticOptimizer, ::ListOfNonstandardBridges{Float64})
    return Type[
        MOI.Bridges.Constraint.SOCtoNonConvexQuadBridge{Float64},
        MOI.Bridges.Constraint.RSOCtoNonConvexQuadBridge{Float64},
    ]
end
```

An optimizer defining an internal bridge

Suppose an optimizer can exploit specific structure of a constraint, e.g., it can exploit the structure of the matrix A in the linear system of equations `A * x = b`.

The optimizer can define the function:

```
struct MatrixAffineFunction{T} <: MOI.AbstractVectorFunction
    A::SomeStructuredMatrixType{T}
    b::Vector{T}
end
```

and then a bridge

```
struct MatrixAffineFunctionBridge{T} <: MOI.Constraint.AbstractBridge
    # ...
end
# ...
```

from `VectorAffineFunction{T}` to the `MatrixAffineFunction`. Finally, it defines:

```
function MOI.get(::Optimizer{T}, ::ListOfNonstandardBridges{T}) where {T}
    return Type[MatrixAffineFunctionBridge{T}]
end
```

source

`MathOptInterface.Bridges.debug_supports_constraint` – Function.

```
debug_supports_constraint(
    b::LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet};
    io::IO = Base.stdout,
)
```

Prints to `io` explanations for the value of `MOI.supports_constraint` with the same arguments.

source

`MathOptInterface.Bridges.debug_supports` – Function.

```
debug_supports(
    b::LazyBridgeOptimizer,
    ::MOI.ObjectiveFunction{F};
    io::IO = Base.stdout,
) where F
```

Prints to `io` explanations for the value of `MOI.supports` with the same arguments.

source

`MathOptInterface.Bridges.bridged_variable_function` – Function.

```
bridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a `MOI.AbstractScalarFunction` of variables of `b.model` that equals `vi`. That is, if the variable `vi` is bridged, it returns its expression in terms of the variables of `b.model`. Otherwise, it returns `vi`.

source

`MathOptInterface.Bridges.unbridged_variable_function` – Function.

```
unbridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a `MOI.AbstractScalarFunction` of variables of `b` that equals `vi`. That is, if the variable `vi` is an internal variable of `b.model` created by a bridge but not visible to the user, it returns its expression in terms of the variables of bridged variables. Otherwise, it returns `vi`.

source

`MathOptInterface.Bridges.bridged_function` – Function.

```
bridged_function(b::AbstractBridgeOptimizer, value)::typeof(value)
```

Substitute any bridged `MOI.VariableIndex` in `value` by an equivalent expression in terms of variables of `b.model`.

source

`MathOptInterface.Bridges.Variable.unbridged_map` – Function.

unbridged_map(bridge::MOI.Bridges.Variable.AbstractBridge, vi::MOI.VariableIndex,)

For a bridged variable in a scalar set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vi`.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vis::Vector{MOI.VariableIndex},
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vis`. If this method is not implemented, it falls back to calling the following method for every variable of `vis`.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vi::MOI.VariableIndex,
    i::MOIB.IndexInVector,
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vi` corresponding to the `i`th variable of the vector.

If there is no way to recover the expression in terms of the bridged variable(s) `vi(s)`, return `nothing`. See `ZerosBridge` for an example of bridge returning `nothing`.

source

Constraint bridges

`MathOptInterface.Bridges.Constraint.AbstractBridge` – Type.

```
AbstractBridge
```

Subtype of `MathOptInterface.Bridges.AbstractBridge` for constraint bridges.

source

`MathOptInterface.Bridges.Constraint.AbstractFunctionConversionBridge` – Type.

```
abstract type AbstractFunctionConversionBridge{F, S} <: AbstractBridge end
```

Bridge a constraint G-in-S into a constraint F-in-S where F and G are equivalent representations of the same function. By convention, the transformed function is stored in the `constraint` field.

source

`MathOptInterface.Bridges.Constraint.SingleBridgeOptimizer` – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <:
AbstractBridgeOptimizer
```

The `SingleBridgeOptimizer` bridges any constraint supported by the bridge BT. This is in contrast with the `MathOptInterface.Bridges.LazyBridgeOptimizer` which only bridges the constraints that are unsupported by the internal model, even if they are supported by one of its bridges.

source

`MathOptInterface.Bridges.Constraint.add_all_bridges` – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the `Bridges.Constraint` submodule to `bridged_model`. The coefficient type used is T.

source

SetMap bridges `MathOptInterface.Bridges.Variable.SetMapBridge` – Type.

```
abstract type SetMapBridge{T,S1,S2} <: AbstractBridge end
```

Consider two type of sets S1, S2 and a linear mapping A that the image of a set of type S1 under A is a set of type S2. A `SetMapBridge{T,S1,S2}` is a bridge that substitutes constrained variables in S2 into the image through A of constrained variables in S1.

The linear map A is described by `MathOptInterface.Bridges.map_set`, `MathOptInterface.Bridges.map_function`. Implementing a method for these two functions is sufficient to bridge constrained variables. In order for the getters and setters of dual solutions, starting values, etc... to work as well a method for the following functions should be implemented as well: `MathOptInterface.Bridges.inverse_map_set`, `MathOptInterface.Bridges.inverse` `MathOptInterface.Bridges.adjoint_map_function` and `MathOptInterface.Bridges.inverse_adjoint_map_function`. See the docstrings of the function to see which feature would be missing it it was not implemented for a given bridge.

source

`MathOptInterface.Bridges.Constraint.SetMapBridge` – Type.

```
abstract type SetMapBridge{T,S2,S1,F,G} <: AbstractBridge end
```

Consider two type of sets S1, S2 and a linear mapping A that the image of a set of type S1 under A is a set of type S2. A `SetMapBridge{T,S2,S1,F,G}` is a bridge that maps G-in-S2 constraints into F-in-S1 by mapping the function through A.

The linear map A is described by `MathOptInterface.Bridges.map_set`, `MathOptInterface.Bridges.map_function`. Implementing a method for these two functions is sufficient to bridge constraints. In order for the getters and setters of dual solutions, starting values, etc... to work as well a method for the following functions should be implemented as well: `MathOptInterface.Bridges.inverse_map_set`, `MathOptInterface.Bridges.inverse_map` `MathOptInterface.Bridges.adjoint_map_function` and `MathOptInterface.Bridges.inverse_adjoint_map_function`. See the docstrings of the function to see which feature would be missing it it was not implemented for a given bridge.

source

`MathOptInterface.Bridges.map_set` – Function.

```
map_set(::Type{BT}, set) where {BT}
```

Return the image of `set` through the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used for bridging the constraint and setting the `MathOptInterface.ConstraintSet`.

source

`MathOptInterface.Bridges.inverse_map_set` – Function.

```
inverse_map_set(::Type{BT}, set) where {BT}
```

Return the preimage of `set` through the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used for getting the `MathOptInterface.ConstraintSet`.

source

`MathOptInterface.Bridges.map_function` – Function.

```
map_function(::Type{BT}, func) where {BT}
```

Return the image of `func` through the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used for getting the `MathOptInterface.ConstraintPrimal` of variable bridges. For constraint bridges, this is used for bridging the constraint, setting the `MathOptInterface.ConstraintFunction` and `MathOptInterface.ConstraintPrimalStart` and modifying the function with `MathOptInterface.modify`.

```
map_function(::Type{BT}, func, i::IndexInVector) where {BT}
```

Return the scalar function at the `i`th index of the vector function that would be returned by `map_function(BT, func)` except that it may compute the `i`th element. This is used by `bridged_function` and for getting the `MathOptInterface.VariablePrimal` and `MathOptInterface.VariablePrimalStart` of variable bridges.

source

`MathOptInterface.Bridges.inverse_map_function` – Function.

```
inverse_map_function(::Type{BT}, func) where {BT}
```

Return the image of `func` through the inverse of the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used by `Variable.unbridged_map` and for setting the `MathOptInterface.VariablePrima` of variable bridges and for getting the `MathOptInterface.ConstraintFunction`, the `MathOptInterface.ConstraintPrimal` and the `MathOptInterface.ConstraintPrimalStart` of constraint bridges.

source

`MathOptInterface.Bridges.adjoint_map_function` – Function.

```
adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of `func` through the adjoint of the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used for getting the `MathOptInterface.ConstraintDual` and `MathOptInterface.ConstraintDualStart` of constraint bridges.

source

`MathOptInterface.Bridges.inverse_adjoint_map_function` – Function.

```
inverse_adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of `func` through the inverse of the adjoint of the linear map A defined in `Variable.SetMapBridge` and `Constraint.SetMapBridge`. This is used for getting the `MathOptInterface.ConstraintDual` of variable bridges and setting the `MathOptInterface.ConstraintDualStart` of constraint bridges.

source

Bridges implemented `MathOptInterface.Bridges.Constraint.FlipSignBridge` – Type.

```
FlipSignBridge{T, S1, S2, F, G}
```

Bridge a G-in-S1 constraint into an F-in-S2 constraint by multiplying the function by `-1` and taking the point reflection of the set across the origin. The flipped F-in-S constraint is stored in the `constraint` field by convention.

source

`MathOptInterface.Bridges.Constraint.AbstractToIntervalBridge` – Type.

```
AbstractToIntervalBridge{T, S1, F}
```

Bridge a F-in-`Interval` constraint into an F-in-`Interval{T}` constraint where we have either:

- `S1 = MOI.GreaterThan{T}`
- `S1 = MOI.LessThan{T}`

The F-in-`Interval{T}` constraint is stored in the `constraint` field by convention.

> **Warning**
> It is required that T be a `AbstractFloat` type because otherwise `typemin` and `typemax` would either be not implemented (e.g. `BigInt`) or would not give infinite value (e.g. `Int`). For this reason, this bridge is only added to `MathOptInterface.Bridges.full_bridge_optimizer`.
> when T is a subtype of `AbstractFloat`.

source

`MathOptInterface.Bridges.Constraint.GreaterToIntervalBridge` – Type.

```
GreaterToIntervalBridge{T, F<:MOI.AbstractScalarFunction} <:
    AbstractToIntervalBridge{T, MOI.GreaterThan{T}, F}
```

Transforms a F-in-GreaterThan{T} constraint into an F-in-`Interval{T}` constraint.

source

`MathOptInterface.Bridges.Constraint.LessToIntervalBridge` – Type.

```
LessToIntervalBridge{T, F<:MOI.AbstractScalarFunction} <:
    AbstractToIntervalBridge{T, MOI.LessThan{T}, F}
```

Transforms a F-in-LessThan{T} constraint into an F-in-`Interval{T}` constraint.

source

`MathOptInterface.Bridges.Constraint.GreaterToLessBridge` – Type.

```
GreaterToLessBridge{
    T,
    F<:MOI.AbstractScalarFunction,
    G<:MOI.AbstractScalarFunction
} <: FlipSignBridge{T, MOI.GreaterThan{T}, MOI.LessThan{T}, F, G}
```

Transforms a G-in-`GreaterThan{T}` constraint into an F-in-`LessThan{T}` constraint.

source

`MathOptInterface.Bridges.Constraint.LessToGreaterBridge` – Type.

```
LessToGreaterBridge{
    T,
    F<:MOI.AbstractScalarFunction,
    G<:MOI.AbstractScalarFunction
} <: FlipSignBridge{T, MOI.LessThan{T}, MOI.GreaterThan{T}, F, G}
```

Transforms a G-in-LessThan{T} constraint into an F-in-GreaterThan{T} constraint.

source

MathOptInterface.Bridges.Constraint.NonnegToNonposBridge – Type.

```
NonnegToNonposBridge{
    T,
    F<:MOI.AbstractVectorFunction,
    G<:MOI.AbstractVectorFunction
} <: FlipSignBridge{T, MOI.Nonnegatives, MOI.Nonpositives, F, G}
```

Transforms a G-in-Nonnegatives constraint into a F-in-Nonpositives constraint.

source

MathOptInterface.Bridges.Constraint.NonposToNonnegBridge – Type.

```
NonposToNonnegBridge{
    T,
    F<:MOI.AbstractVectorFunction,
    G<:MOI.AbstractVectorFunction,
} <: FlipSignBridge{T, MOI.Nonpositives, MOI.Nonnegatives, F, G}
```

Transforms a G-in-Nonpositives constraint into a F-in-Nonnegatives constraint.

source

MathOptInterface.Bridges.Constraint.VectorizeBridge – Type.

```
VectorizeBridge{T,F,S,G}
```

Transforms a constraint G-in-scalar_set_type(S, T) where S <: VectorLinearSet to F-in-S.

Examples

The constraint VariableIndex-in-LessThan{Float64} becomes VectorAffineFunction{Float64}-in-Nonpositives, where T = Float64, F = VectorAffineFunction{Float64}, S = Nonpositives, and G = VariableIndex.

source

MathOptInterface.Bridges.Constraint.ScalarizeBridge – Type.

```
ScalarizeBridge{T, F, S}
```

Transforms a constraint AbstractVectorFunction-in-vector_set_type(S) where S <: LPCone{T} to F-in-S.

source

MathOptInterface.Bridges.Constraint.ScalarSlackBridge – Type.

```
ScalarSlackBridge{T, F, S}
```

The `ScalarSlackBridge` converts a constraint `G`-in-`S` where `G` is a function different from `VariableIndex` into the constraints `F`-in-`EqualTo{T}` and `VariableIndex`-in-`S`.

`F` is the result of subtracting a `VariableIndex` from `G`. Typically `G` is the same as `F`, but that is not mandatory.

source

`MathOptInterface.Bridges.Constraint.VectorSlackBridge` – Type.

```
VectorSlackBridge{T, F, S}
```

The `VectorSlackBridge` converts a constraint `G`-in-`S` where `G` is a function different from `VectorOfVariables` into the constraints `F`in-`Zeros` and `VectorOfVariables`-in-`S`.

`F` is the result of subtracting a `VectorOfVariables` from `G`. Typically `G` is the same as `F`, but that is not mandatory.

source

`MathOptInterface.Bridges.Constraint.ScalarFunctionizeBridge` – Type.

```
ScalarFunctionizeBridge{T, S}
```

The `ScalarFunctionizeBridge` converts a constraint `VariableIndex`-in-`S` into the constraint `ScalarAffineFunction{T}`-in-`S`.

source

`MathOptInterface.Bridges.Constraint.VectorFunctionizeBridge` – Type.

```
VectorFunctionizeBridge{T, S}
```

The `VectorFunctionizeBridge` converts a constraint `VectorOfVariables`-in-`S` into the constraint `VectorAffineFunction{T}`-in-`S`.

source

`MathOptInterface.Bridges.Constraint.SplitIntervalBridge` – Type.

```
SplitIntervalBridge{T, F, S, LS, US}
```

The `SplitIntervalBridge` splits a `F`-in-`S` constraint into a `F`-in-`LS` and a `F`-in-`US` constraint where we have either:

- `S = MOI.Interval{T}`, `LS = MOI.GreaterThan{T}` and `US = MOI.LessThan{T}`,
- `S = MOI.EqualTo{T}`, `LS = MOI.GreaterThan{T}` and `US = MOI.LessThan{T}`, or
- `S = MOI.Zeros`, `LS = MOI.Nonnegatives` and `US = MOI.Nonpositives`.

For instance, if `F` is `MOI.ScalarAffineFunction` and `S` is `MOI.Interval`, it transforms the constraint $la, x+u$ into the constraints $a, x+l$ and $a, x+u$.

> **Note**
> If `T<:AbstractFloat` and `S` is `MOI.Interval{T}` then no lower (resp. upper) bound constraint is created if the lower (resp. upper) bound is `typemin(T)` (resp. `typemax(T)`). Similarly, when `MathOptInterface.ConstraintSet` is set, a lower or upper bound constraint may be deleted or created accordingly.

source

MathOptInterface.Bridges.Constraint.SOCtoRSOCBridge – Type.

```
SOCtoRSOCBridge{T, F, G}
```

We simply do the inverse transformation of RSOCtoSOCBridge. In fact, as the transformation is an involution, we do the same transformation.

source

MathOptInterface.Bridges.Constraint.RSOCtoSOCBridge – Type.

```
RSOCtoSOCBridge{T, F, G}
```

The `RotatedSecondOrderCone` is `SecondOrderCone` representable; see [BN01, p. 104]. Indeed, we have $2tu = (t/\sqrt{2} + u/\sqrt{2})^2 - (t/\sqrt{2} - u/\sqrt{2})^2$ hence

$$2tu \geq \|x\|_2^2$$

is equivalent to

$$(t/\sqrt{2} + u/\sqrt{2})^2 \geq \|x\|_2^2 + (t/\sqrt{2} - u/\sqrt{2})^2.$$

We can therefore use the transformation $(t, u, x) \mapsto (t/\sqrt{2} + u/\sqrt{2}, t/\sqrt{2} - u/\sqrt{2}, x)$. Note that the linear transformation is a symmetric involution (i.e. it is its own transpose and its own inverse). That means in particular that the norm of constraint primal and dual values are preserved by the tranformation.

[BN01] Ben-Tal, Aharon, and Nemirovski, Arkadi. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

source

MathOptInterface.Bridges.Constraint.SOCtoNonConvexQuadBridge – Type.

```
SOCtoNonConvexQuadBridge{T}
```

Constraints of the form `VectorOfVariables`-in-`SecondOrderCone` can be transformed into a `ScalarQuadraticFunction`-in-`LessThan` and a `ScalarAffineFunction`-in-`GreaterThan`. Indeed, the definition of the second-order cone

$$t \geq \|x\|_2 \ (1)$$

is equivalent to

$$\sum x_i^2 \leq t^2 (2)$$

with $t \geq 0$. (3)

Warning

This transformation starts from a convex constraint (1) and creates a non-convex constraint (2), because the Q matrix associated with the constraint (2) has one negative eigenvalue. This might be wrongly interpreted by a solver. Some solvers can look at (2) and understand that it is a second order cone, but this is not a general rule. For these reasons this bridge is not automatically added by `MOI.Bridges.full_bridge_optimizer`. Care is recommended when adding this bridge to a optimizer.

source

`MathOptInterface.Bridges.Constraint.RSOCtoNonConvexQuadBridge` – Type.

```
RSOCtoNonConvexQuadBridge{T}
```

Constraints of the form `VectorOfVariables`-in-`SecondOrderCone` can be transformed into a `ScalarQuadraticFunction`-in-`LessThan` and a `ScalarAffineFunction`-in-`GreaterThan`. Indeed, the definition of the second-order cone

$$2tu \geq \|x\|_2^2, t, u \geq 0(1)$$

is equivalent to

$$\sum x_i^2 \leq 2tu(2)$$

with $t, u \geq 0.$ (3)

WARNING This transformation starts from a convex constraint (1) and creates a non-convex constraint (2), because the Q matrix associated with the constraint 2 has two negative eigenvalues. This might be wrongly interpreted by a solver. Some solvers can look at (2) and understand that it is a rotated second order cone, but this is not a general rule. For these reasons, this bridge is not automatically added by `MOI.Bridges.full_bridge_optimizer`. Care is recommended when adding this bridge to an optimizer.

source

`MathOptInterface.Bridges.Constraint.QuadtoSOCBridge` – Type.

```
QuadtoSOCBridge{T}
```

The set of points x satisfying the constraint

$$\frac{1}{2}x^TQx + a^Tx + b \leq 0$$

is a convex set if `Q` is positive semidefinite and is the union of two convex cones if `a` and `b` are zero (i.e. homogeneous case) and `Q` has only one negative eigenvalue. Currently, only the non-homogeneous transformation is implemented, see the Note section below for more details.

Non-homogeneous case

If `Q` is positive semidefinite, there exists `U` such that $Q = U^TU$, the inequality can then be rewritten as

$$\|Ux\|_2^2 \leq 2(-a^Tx - b)$$

which is equivalent to the membership of `(1, -a^T x - b, Ux)` to the rotated second-order cone.

Homogeneous case

If `Q` has only one negative eigenvalue, the set of `x` such that $x^T Q x \leq 0$ is the union of a convex cone and its opposite. We can choose which one to model by checking the existence of bounds on variables as shown below.

Second-order cone

If `Q` is diagonal and has eigenvalues `(1, 1, -1)`, the inequality $x^2 + x^2 \leq z^2$ combined with $z \geq 0$ defines the Lorenz cone (i.e. the second-order cone) but when combined with $z \leq 0$, it gives the opposite of the second order cone. Therefore, we need to check if the variable `z` has a lower bound 0 or an upper bound 0 in order to determine which cone is

Rotated second-order cone

The matrix `Q` corresponding to the inequality $x^2 \leq 2yz$ has one eigenvalue 1 with eigenvectors `(1, 0, 0)` and `(0, 1, -1)` and one eigenvalue `-1` corresponding to the eigenvector `(0, 1, 1)`. Hence if we intersect this union of two convex cone with the halfspace $x + y \geq 0$, we get the rotated second-order cone and if we intersect it with the halfspace $x + y \leq 0$ we get the opposite of the rotated second-order cone. Note that `y` and `z` have the same sign since `yz` is nonnegative hence $x + y \geq 0$ is equivalent to $x \geq 0$ and $y \geq 0$.

Note

The check for existence of bound can be implemented (but inefficiently) with the current interface but if bound is removed or transformed (e.g. `≤ 0` transformed into `≥ 0`) then the bridge is no longer valid. For this reason the homogeneous version of the bridge is not implemented yet.

source

`MathOptInterface.Bridges.Constraint.SOCtoPSDBridge` – Type.

The `SOCtoPSDBridge` transforms the second order cone constraint $\|x\| \leq t$ into the semidefinite cone constraints

$$\begin{pmatrix} t & x^\top \\ x & tI \end{pmatrix} \succeq 0$$

Indeed by the Schur Complement, it is positive definite iff

$$\begin{aligned} tI &\succ 0 \\ t - x^\top (tI)^{-1} x &\succ 0 \end{aligned}$$

which is equivalent to

$$\begin{aligned} t &> 0 \\ t^2 &> x^\top x \end{aligned}$$

Warning

This bridge is not added by default by `MOI.Bridges.full_bridge_optimizer` as bridging second order cone constraints to semidefinite constraints can be achieved by the `SOCtoRSOCBridge` followed by the `RSOCtoPSDBridge` while creating a smaller semidefinite constraint.

source

MathOptInterface.Bridges.Constraint.RSOCtoPSDBridge – Type.

The `RSOCtoPSDBridge` transforms the second order cone constraint $\|x\| \le 2tu$ with $u \ge 0$ into the semidefinite cone constraints

$$\begin{pmatrix} t & x^\top \\ x & 2uI \end{pmatrix} \succeq 0$$

Indeed by the Schur Complement, it is positive definite iff

$$\begin{aligned} uI &\succ 0 \\ t - x^\top (2uI)^{-1} x &\succ 0 \end{aligned}$$

which is equivalent to

$$\begin{aligned} u &> 0 \\ 2tu &> x^\top x \end{aligned}$$

source

MathOptInterface.Bridges.Constraint.NormInfinityBridge – Type.

```
NormInfinityBridge{T}
```

The `NormInfinityCone` is representable with LP constraints, since $t \ge \max_i |x_i|$ if and only if $t \ge x_i$ and $t \ge -x_i$ for all i.

source

MathOptInterface.Bridges.Constraint.NormOneBridge – Type.

```
NormOneBridge{T}
```

The `NormOneCone` is representable with LP constraints, since $t \ge \sum_i |x_i|$ if and only if there exists a vector y such that $t \ge \sum_i y_i$ and $y_i \ge x_i$, $y_i \ge -x_i$ for all i.

source

MathOptInterface.Bridges.Constraint.GeoMeantoRelEntrBridge – Type.

```
GeoMeantoRelEntrBridge{T}
```

The `geometric mean` cone is representable with a relative entropy constraint and a nonnegative auxiliary variable.

This is because $u \le \prod_{i=1}^n w_i^{1/n}$ is equivalent to $y \ge 0$ and $0 \le u + y \le \prod_{i=1}^n w_i^{1/n}$, and the latter inequality is equivalent to $1 \le \prod_{i=1}^n (\frac{w_i}{u+y})^{1/n}$, which is equivalent to $0 \le \sum_{i=1}^n \log(\frac{w_i}{u+y})^{1/n}$, which is equivalent to $0 \ge \sum_{i=1}^n (u+y) \log(\frac{u+y}{w_i})$.

Thus $(u, w) \in GeometricMeanCone(1+n)$ is representable as $y \ge 0$, $(0, w, (u+y)e) \in RelativeEntropyCone(1+2n)$, where e is a vector of ones.

source

`MathOptInterface.Bridges.Constraint.GeoMeanBridge` – Type.

```
GeoMeanBridge{T, F, G, H}
```

The `GeometricMeanCone` is `SecondOrderCone` representable; see [1, p. 105].

The reformulation is best described in an example.

Consider the cone of dimension 4:

$$t \le \sqrt[3]{x_1 x_2 x_3}$$

This can be rewritten as $\exists x_{21} \ge 0$ such that:

$$\begin{aligned} t &\le x_{21}, \\ x_{21}^4 &\le x_1 x_2 x_3 x_{21}. \end{aligned}$$

Note that we need to create x_{21} and not use t^4 directly as t is allowed to be negative. Now, this is equivalent to:

$$\begin{aligned} t &\le x_{21}/\sqrt{4}, \\ x_{21}^2 &\le 2x_{11}x_{12}, \\ x_{11}^2 &\le 2x_1x_2, & x_{12}^2 &\le 2x_3(x_{21}/\sqrt{4}). \end{aligned}$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

source

`MathOptInterface.Bridges.Constraint.RelativeEntropyBridge` – Type.

```
RelativeEntropyBridge{T}
```

The `RelativeEntropyCone` is representable with exponential cone and LP constraints, since $u \ge \sum_{i=1}^n w_i \log(\frac{w_i}{v_i})$ if and only if there exists a vector y such that $u \ge \sum_i y_i$ and $y_i \ge w_i \log(\frac{w_i}{v_i})$ or equivalently $v_i \ge w_i \exp(\frac{-y_i}{w_i})$ or equivalently $(-y_i, w_i, v_i) \in ExponentialCone$, for all i.

source

`MathOptInterface.Bridges.Constraint.NormSpectralBridge` – Type.

```
NormSpectralBridge{T}
```

The `NormSpectralCone` is representable with a PSD constraint, since $t \ge \sigma_1(X)$ if and only if $[tI X^\top; X tI] \succ 0$.

source

`MathOptInterface.Bridges.Constraint.NormNuclearBridge` – Type.

```
NormNuclearBridge{T}
```

The NormNuclearCone is representable with an SDP constraint and extra variables, since $t \geq \sum_i \sigma_i(X)$ if and only if there exists symmetric matrices U, V such that $[UX^\top; XV] \succ 0$ and $t \geq (tr(U) + tr(V))/2$.

source

MathOptInterface.Bridges.Constraint.SquareBridge – Type.

```
SquareBridge{T, F<:MOI.AbstractVectorFunction,
             G<:MOI.AbstractScalarFunction,
             TT<:MOI.AbstractSymmetricMatrixSetTriangle,
             ST<:MOI.AbstractSymmetricMatrixSetSquare} <: AbstractBridge
```

The SquareBridge reformulates the constraint of a square matrix to be in ST to a list of equality constraints for pair or off-diagonal entries with different expressions and a TT constraint the upper triangular part of the matrix.

For instance, the constraint for the matrix

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ 1+x & 2+x & 3-x \\ 2-3x & 2+x & 2x \end{pmatrix}$$

to be PSD can be broken down to the constraint of the symmetric matrix

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ \cdot & 2+x & 3-x \\ \cdot & \cdot & 2x \end{pmatrix}$$

and the equality constraint between the off-diagonal entries (2, 3) and (3, 2) $2x == 1$. Note that now symmetrization constraint need to be added between the off-diagonal entries (1, 2) and (2, 1) or between (1, 3) and (3, 1) since the expressions are the same.

source

MathOptInterface.Bridges.Constraint.RootDetBridge – Type.

```
RootDetBridge{T,F,G,H}
```

The RootDetConeTriangle is representable by a PositiveSemidefiniteConeTriangle and an GeometricMeanCone constraints; see [1, p. 149].

Indeed, $t \leq \det(X)^{1/n}$ if and only if there exists a lower triangular matrix such that:

$$\begin{pmatrix} X & \\ \top & \mathrm{Diag}() \end{pmatrix} \succeq 0$$
$$t \leq ({}_{11}{}_{22} \cdots {}_{nn})^{1/n}$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

source

MathOptInterface.Bridges.Constraint.LogDetBridge – Type.

```
LogDetBridge{T,F,G,H,I}
```

The `LogDetConeTriangle` is representable by a `PositiveSemidefiniteConeTriangle` and `ExponentialCone` constraints.

Indeed, $\log\det(X) = \log(\delta_1) + \cdots + \log(\delta_n)$ where δ_1, ..., δ_n are the eigenvalues of X.

Adapting the method from [1, p. 149], we see that $t \leq u \log(\det(X/u))$ for $u > 0$ if and only if there exists a lower triangular matrix such that

$$\begin{pmatrix} X & \\ ^\top & \mathrm{Diag}() \end{pmatrix} \succeq 0$$

$$t \leq u \log(_{11}/u) + u \log(_{22}/u) + \cdots + u \log(_{nn}/u)$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001. "'

source

`MathOptInterface.Bridges.Constraint.IndicatorActiveOnFalseBridge` – Type.

```
IndicatorActiveOnFalseBridge{T}
```

The `IndicatorActiveOnFalseBridge` replaces an indicator constraint activated on 0 with a variable z_0 with the constraint activated on 1, with a variable z_1. It stores the added `variable` and added constraints:

- $z_1 \in \mathbb{B}$ in `zero_one_cons`
- $z_0 + z_1 == 1$ in 'indisjunction_cons'
- The added `ACTIVATE_ON_ONE` indicator constraint in `indicator_cons_index`.

source

`MathOptInterface.Bridges.Constraint.IndicatorSOS1Bridge` – Type.

```
IndicatorSOS1Bridge{T,S<:MOI.AbstractScalarSet}
```

The `IndicatorSOS1Bridge` replaces an indicator constraint of the following form: $z \in \mathbb{B}, z == 1 \implies f(x) \in S$ with a SOS1 constraint: $z \in \mathbb{B}, slack$ free, $f(x) + slack \in S, SOS1(slack, z)$.

source

`MathOptInterface.Bridges.Constraint.SemiToBinaryBridge` – Type.

```
SemiToBinaryBridge{T, S <: MOI.AbstractScalarSet}
```

The `SemiToBinaryBridge` replaces a Semicontinuous constraint: $x \in$ Semicontinuous(l, u) is replaced by: $z \in \{0, 1\}$, $x \leq z \cdot u$, $x \geq z \cdot l$.

The `SemiToBinaryBridge` replaces a Semiinteger constraint: $x \in Semiinteger(l, u)$ is replaced by: $z \in \{0, 1\}$, $x \in \mathbb{Z}$, $x \leq z \cdot u$, $x \geq z \cdot l$.

source

`MathOptInterface.Bridges.Constraint.ZeroOneBridge` – Type.

```
ZeroOneBridge{T}
```

The `ZeroOneBridge` splits a `MOI.VariableIndex`-in-`MOI.ZeroOne` constraint into a `MOI.VariableIndex`-in-`MOI.Integer` constraint and a `MOI.VariableIndex`-in-`MOI.Interval(0, 1)` constraint.

source

Variable bridges

MathOptInterface.Bridges.Variable.AbstractBridge – Type.

```
AbstractBridge
```

Subtype of MathOptInterface.Bridges.AbstractBridge for variable bridges.

source

MathOptInterface.Bridges.Variable.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <:
AbstractBridgeOptimizer
```

The SingleBridgeOptimizer bridges any constrained variables supported by the bridge BT. This is in contrast with the MathOptInterface.Bridges.LazyBridgeOptimizer which only bridges the constrained variables that are unsupported by the internal model, even if they are supported by one of its bridges.

> **Note**
> Two bridge optimizers using variable bridges cannot be used together as both of them assume that the underlying model only returns variable indices with nonnegative values.

source

MathOptInterface.Bridges.Variable.add_all_bridges – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the Bridges.Variable submodule to bridged_model. The coefficient type used is T.

source

Bridges implemented MathOptInterface.Bridges.Variable.FlipSignBridge – Type.

```
FlipSignBridge{T, S1, S2}
```

Bridge constrained variables in S1 into constrained variables in S2 by multiplying the variables by -1 and taking the point reflection of the set across the origin. The flipped MOI.VectorOfVariables-in-S constraint is stored in the flipped_constraint field by convention.

source

MathOptInterface.Bridges.Variable.ZerosBridge – Type.

```
ZerosBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in MathOptInterface.Zeros to zeros, which ends up creating no variables in the underlying model.

The bridged variables are therefore similar to parameters with zero values. Parameters with non-zero value can be created with constrained variables in MOI.EqualTo by combining a VectorizeBridge and this bridge. The functions cannot be unbridged, given a function, we cannot determine, if the bridged variables were used.

The dual values cannot be determined by the bridge but they can be determined by the bridged optimizer using MathOptInterface.Utilities.get_fallback if a CachingOptimizer is used (since ConstraintFunction cannot be got as functions cannot be unbridged).

source

MathOptInterface.Bridges.Variable.FreeBridge – Type.

```
FreeBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in MOI.Reals to the difference of constrained variables in MOI.Nonnegatives.

source

MathOptInterface.Bridges.Variable.NonposToNonnegBridge – Type.

```
NonposToNonnegBridge{T} <:
    FlipSignBridge{T, MOI.Nonpositives, MOI.Nonnegatives}
```

Transforms constrained variables in Nonpositives into constrained variables in Nonnegatives.

source

MathOptInterface.Bridges.Variable.VectorizeBridge – Type.

```
VectorizeBridge{T, S}
```

Transforms a constrained variable in scalar_set_type(S, T) where S <: VectorLinearSet into a constrained vector of one variable in S. For instance, VectorizeBridge{Float64, MOI.Nonnegatives} transforms a constrained variable in MOI.GreaterThan{Float64} into a constrained vector of one variable in MOI.Nonnegatives.

source

MathOptInterface.Bridges.Variable.SOCtoRSOCBridge – Type.

```
SOCtoRSOCBridge{T} <:
↪   Bridges.Variable.SetMapBridge{T,MOI.RotatedSecondOrderCone,MOI.SecondOrderCone}
```

Same transformation as MOI.Bridges.Constraint.SOCtoRSOCBridge.

source

MathOptInterface.Bridges.Variable.RSOCtoSOCBridge – Type.

```
RSOCtoSOCBridge{T} <:
↪   Bridges.Variable.SetMapBridge{T,MOI.SecondOrderCone,MOI.RotatedSecondOrderCone}
```

Same transformation as MOI.Bridges.Constraint.RSOCtoSOCBridge.

source

MathOptInterface.Bridges.Variable.RSOCtoPSDBridge – Type.

```
RSOCtoPSDBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in MathOptInterface.RotatedSecondOrderCone to constrained variables in MathOptInterface.PositiveSemidefiniteConeTriangle.

source

Objective bridges

MathOptInterface.Bridges.Objective.AbstractBridge – Type.

```
AbstractBridge
```

Subtype of MathOptInterface.Bridges.AbstractBridge for objective bridges.

source

MathOptInterface.Bridges.Objective.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <: AbstractBridgeOptimizer
```

The `SingleBridgeOptimizer` bridges any objective functions supported by the bridge BT. This is in contrast with the MathOptInterface.Bridges.LazyBridgeOptimizer which only bridges the objective functions that are unsupported by the internal model, even if they are supported by one of its bridges.

source

MathOptInterface.Bridges.Objective.add_all_bridges – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the `Bridges.Objective` submodule to `bridged_model`. The coefficient type used is T.

source

Bridges implemented MathOptInterface.Bridges.Objective.SlackBridge – Type.

```
SlackBridge{T, F, G}
```

The `SlackBridge` converts an objective function of type G into a MOI.VariableIndex objective by creating a slack variable and a F-in-MOI.LessThan constraint for minimization or F-in-MOI.LessThan constraint for maximization where F is `MOI.Utilities.promote_operation(-, T, G, MOI.VariableIndex}`. Note that when using this bridge, changing the optimization sense is not supported. Set the sense to `MOI.FEASIBILITY_SENSE` first to delete the bridge in order to change the sense, then re-add the objective.

source

MathOptInterface.Bridges.Objective.FunctionizeBridge – Type.

```
FunctionizeBridge{T}
```

The `FunctionizeBridge` converts a `VariableIndex` objective into a `ScalarAffineFunction{T}` objective.

source

Chapter 28

FileFormats

28.1 Overview

The FileFormats submodule

The `FileFormats` module provides functionality for reading and writing MOI models using `write_to_file` and `read_from_file`.

Supported file types

You must read and write files to a `FileFormats.Model` object. Specifc the file-type by passing a `FileFormats.FileFormat` enum. For example:

The Conic Benchmark Format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF)
A Conic Benchmark Format (CBF) model
```

The LP file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_LP)
A .LP-file model
```

The MathOptFormat file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model
```

The MPS file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model
```

The NL file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_NL)
An AMPL (.nl) model
```

The SDPA file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_SDPA)
A SemiDefinite Programming Algorithm Format (SDPA) model
```

Write to file

To write a model `src` to a MathOptFormat file, use:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> MOI.add_variable(src)
MathOptInterface.VariableIndex(1)

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap with 1 entry:
  VariableIndex(1) => VariableIndex(1)

julia> MOI.write_to_file(dest, "file.mof.json")

julia> print(read("file.mof.json", String))
{
  "name": "MathOptFormat Model",
  "version": {
    "major": 1,
    "minor": 0
  },
  "variables": [
    {
      "name": "x1"
    }
  ],
  "objective": {
    "sense": "feasibility"
  },
  "constraints": []
}
```

Read from file

To read a MathOptFormat file, use:

```
julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model

julia> MOI.read_from_file(dest, "file.mof.json")

julia> MOI.get(dest, MOI.ListOfVariableIndices())
1-element Vector{MathOptInterface.VariableIndex}:
 MathOptInterface.VariableIndex(1)

julia> rm("file.mof.json")  # Clean up after ourselves.
```

Detecing the filetype automatically

Instead of the `format` keyword, you can also use the `filename` keyword argument to `FileFormats.Model`. This will attempt to automatically guess the format from the file extension. For example:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> MOI.write_to_file(dest, "file.cbf.gz")

julia> src_2 = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> MOI.write_to_file(dest, "file.cbf.gz")

julia> src_2 = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.read_from_file(src_2, "file.cbf.gz")

julia> rm("file.cbf.gz")  # Clean up after ourselves.
```

Note how the compression format (GZip) is also automatically detected from the filename.

Unsupported constraints

In some cases `src` may contain constraints that are not supported by the file format (e.g., the CBF format supports integer variables but not binary). If so, copy `src` to a bridged model using `Bridges.full_bridge_optimizer`:

```
src = MOI.Utilities.Model{Float64}()
x = MOI.add_variable(model)
MOI.add_constraint(model, x, MOI.ZeroOne())
dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF)
bridged = MOI.Bridges.full_bridge_optimizer(dest, Float64)
MOI.copy_to(bridged, src)
MOI.write_to_file(dest, "my_model.cbf")
```

> **Note**
>
> Even after bridging, it may still not be possible to write the model to file because of unsupported constraints (e.g., PSD variables in the LP file format).

Read and write to io

In addition to write_to_file and read_from_file, you can read and write directly from IO streams using Base.write and Base.read!:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> io = IOBuffer();

julia> write(io, dest)

julia> seekstart(io);

julia> src_2 = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model

julia> read!(io, src_2);
```

Validating MOF files

MathOptFormat files are governed by a schema. Use JSONSchema.jl to check if a .mof.json file satisfies the schema.

First, construct the schema object as follows:

```
julia> import JSON, JSONSchema

julia> schema = JSONSchema.Schema(JSON.parsefile(MOI.FileFormats.MOF.SCHEMA_PATH))
A JSONSchema
```

Then, check if a model file is valid using isvalid:

```
julia> good_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 0
         },
         "variables": [{"name": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(good_model, schema)
true
```

If we construct an invalid file, for example by mis-typing name as NaMe, the validation fails:

```
julia> bad_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 0
         },
         "variables": [{"NaMe": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(bad_model, schema)
false
```

Use `JSONSchema.validate` to obtain more insight into why the validation failed:

```
julia> JSONSchema.validate(bad_model, schema)
Validation failed:
path:         [variables][1]
instance:     Dict{String, Any}("NaMe" => "x")
schema key:   required
schema value: Any["name"]
```

28.2 API Reference

File Formats

Functions to help read and write MOI models to/from various file formats. See The FileFormats submodule for more details.

`MathOptInterface.FileFormats.Model` – Function.

```
Model(
    ;
    format::FileFormat = FORMAT_AUTOMATIC,
    filename::Union{Nothing, String} = nothing,
    kwargs...
)
```

Return model corresponding to the `FileFormat` `format`, or, if `format == FORMAT_AUTOMATIC`, guess the format from `filename`.

The `filename` argument is only needed if `format == FORMAT_AUTOMATIC`.

`kwargs` are passed to the underlying model constructor.

source

`MathOptInterface.FileFormats.FileFormat` – Type.

```
FileFormat
```

List of accepted export formats.

- `FORMAT_AUTOMATIC`: try to detect the file format based on the file name

- `FORMAT_CBF`: the Conic Benchmark format
- `FORMAT_LP`: the LP file format
- `FORMAT_MOF`: the MathOptFormat file format
- `FORMAT_MPS`: the MPS file format
- `FORMAT_NL`: the AMPL .nl file format
- `FORMAT_SDPA`: the SemiDefinite Programming Algorithm format

source

Chapter 29

Utilities

29.1 Overview

The Utilities submodule

The Utilities submodule provides a variety of functionality for managing `MOI.ModelLike` objects.

Utilities.Model

`Utilities.Model` provides an implementation of a `ModelLike` that efficiently supports all functions and sets defined within MOI. However, given the extensibility of MOI, this might not cover all use cases.

Create a model as follows:

```
julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
```

Utilities.UniversalFallback

`Utilities.UniversalFallback` is a layer that sits on top of any `ModelLike` and provides non-specialized (slower) fallbacks for constraints and attributes that the underlying `ModelLike` does not support.

For example, `Utilities.Model` doesn't support some variable attributes like `VariablePrimalStart`, so JuMP uses a combination of Universal fallback and `Utilities.Model` as a generic problem cache:

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
MOIU.UniversalFallback{MOIU.Model{Float64}}
fallback for MOIU.Model{Float64}
```

> **Warning**
>
> Adding a UniversalFallback means that your `model` will now support all constraints, even if the inner-model does not! This can lead to unexpected behavior.

Utilities.@model

For advanced use cases that need efficient support for functions and sets defined outside of MOI (but still known at compile time), we provide the `Utilities.@model` macro.

The `@model` macro takes a name (for a new type, which must not exist yet), eight tuples specifying the types of constraints that are supported, and then a `Bool` indicating the type is a subtype of `MOI.AbstractOptimizer` (if `true`) or `MOI.ModelLike` (if `false`).

The eight tuples are in the following order:

1. Un-typed scalar sets, e.g., `Integer`
2. Typed scalar sets, e.g., `LessThan`
3. Un-typed vector sets, e.g., `Nonnegatives`
4. Typed vector sets, e.g., `PowerCone`
5. Un-typed scalar functions, e.g., `VariableIndex`
6. Typed scalar functions, e.g., `ScalarAffineFunction`
7. Un-typed vector functions, e.g., `VectorOfVariables`
8. Typed vector functions, e.g., `VectorAffineFunction`

The tuples can contain more than one element. Typed-sets must be specified without their type parameter, i.e., `MOI.LessThan`, not `MOI.LessThan{Float64}`.

Here is an example:

```
julia> MOI.Utilities.@model(
           MyNewModel,
           (MOI.Integer,),                  # Un-typed scalar sets
           (MOI.GreaterThan,),              # Typed scalar sets
           (MOI.Nonnegatives,),             # Un-typed vector sets
           (MOI.PowerCone,),                # Typed vector sets
           (MOI.VariableIndex,),            # Un-typed scalar functions
           (MOI.ScalarAffineFunction,),     # Typed scalar functions
           (MOI.VectorOfVariables,),        # Un-typed vector functions
           (MOI.VectorAffineFunction,),     # Typed vector functions
           true,                            # <:MOI.AbstractOptimizer?
       )
MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T},
↪  MathOptInterface.Utilities.VariablesContainer{T}, MyNewModelFunctionConstraints{T}} where T

julia> model = MyNewModel{Float64}()
MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64}, MOIU.VariablesContainer{Float64},
↪  MyNewModelFunctionConstraints{Float64}}
```

Warning

`MyNewModel` supports every `VariableIndex`-in-Set constraint, as well as `VariableIndex`, `ScalarAffineFunction`, and `ScalarQuadraticFunction` objective functions. Implement `MOI.supports` as needed to forbid constraint and objective function combinations.

As another example, PATHSolver, which only supports `VectorAffineFunction`-in-`Complements` defines its optimizer as:

```
julia> MOI.Utilities.@model(
           PathOptimizer,
           (),  # Scalar sets
           (),  # Typed scalar sets
           (MOI.Complements,),  # Vector sets
```

```
           (),  # Typed vector sets
           (),  # Scalar functions
           (),  # Typed scalar functions
           (),  # Vector functions
           (MOI.VectorAffineFunction,),  # Typed vector functions
           true,  # is_optimizer
       )
MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T},
↪  MathOptInterface.Utilities.VariablesContainer{T},
↪  MathOptInterface.Utilities.VectorOfConstraints{MathOptInterface.VectorAffineFunction{T},
↪  MathOptInterface.Complements}} where T
```

However, `PathOptimizer` does not support some `VariableIndex`-in-Set constraints, so we must explicitly define:

```
julia> function MOI.supports_constraint(
           ::PathOptimizer,
           ::Type{MOI.VariableIndex},
           ::Type{Union{<:MOI.Semiinteger,MOI.Semicontinuous,MOI.ZeroOne,MOI.Integer}}
       )
           return false
       end
```

Finally, PATH doesn't support an objective function, so we need to add:

```
julia> MOI.supports(::PathOptimizer, ::MOI.ObjectiveFunction) = false
```

> **Warning**
>
> This macro creates a new type, so it must be called from the top-level of a module, e.g., it cannot be called from inside a function.

Utilities.CachingOptimizer

A [`Utilities.CachingOptimizer`] is an MOI layer that abstracts the difference between solvers that support incremental modification (e.g., they support adding variables one-by-one), and solvers that require the entire problem in a single API call (e.g., they only accept the A, b and c matrices of a linear program).

It has two parts:

1. A cache, where the model can be built and modified incrementally
2. An optimizer, which is used to solve the problem

```
julia> model = MOI.Utilities.CachingOptimizer(
           MOI.Utilities.Model{Float64}(),
           PathOptimizer{Float64}(),
       )
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
```

```
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

A `Utilities.CachingOptimizer` may be in one of three possible states:

- `NO_OPTIMIZER`: The CachingOptimizer does not have any optimizer.
- `EMPTY_OPTIMIZER`: The CachingOptimizer has an empty optimizer, and it is not synchronized with the cached model. Modifications are forwarded to the cache, but not to the optimizer.
- `ATTACHED_OPTIMIZER`: The CachingOptimizer has an optimizer, and it is synchronized with the cached model. Modifications are forwarded to the optimizer. If the optimizer does not support modifications, and error will be thrown.

Use `Utilities.attach_optimizer` to go from `EMPTY_OPTIMIZER` to `ATTACHED_OPTIMIZER`:

```
julia> MOI.Utilities.attach_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state ATTACHED_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Info

You must be in `ATTACHED_OPTIMIZER` to use `optimize!`.

Use `Utilities.reset_optimizer` to go from `ATTACHED_OPTIMIZER` to `EMPTY_OPTIMIZER`:

```
julia> MOI.Utilities.reset_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Info

Calling `MOI.empty!(model)` also resets the state to `EMPTY_OPTIMIZER`. So after emptying a model, the modification will only be applied to the cache.

Use `Utilities.drop_optimizer` to go from any state to NO_OPTIMIZER:

```
julia> MOI.Utilities.drop_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state NO_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer nothing
```

Pass an empty optimizer to `Utilities.reset_optimizer` to go from NO_OPTIMIZER to EMPTY_OPTIMIZER:

```
julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Deciding when to attach and reset the optimizer is tedious, and you will often write code like this:

```
try
    # modification
catch
    MOI.Utilities.reset_optimizer(model)
    # Re-try modification
end
```

To make this easier, `Utilities.CachingOptimizer` has two modes of operation:

- `AUTOMATIC`: The `CachingOptimizer` changes its state when necessary. Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to `EMPTY_OPTIMIZER` mode.
- `MANUAL`: The user must change the state of the `CachingOptimizer`. Attempting to perform an operation in the incorrect state results in an error.

By default, `AUTOMATIC` mode is chosen. However, you can create a `CachingOptimizer` in `MANUAL` mode as follows:

```
julia> model = MOI.Utilities.CachingOptimizer(
           MOI.Utilities.Model{Float64}(),
           MOI.Utilities.MANUAL,
       )
```

```
MOIU.CachingOptimizer{MOI.AbstractOptimizer, MOIU.Model{Float64}}
in state NO_OPTIMIZER
in mode MANUAL
with model cache MOIU.Model{Float64}
with optimizer nothing

julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> model
MOIU.CachingOptimizer{MOI.AbstractOptimizer, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode MANUAL
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Printing

Use `print` to print the formulation of the model.

```
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MathOptInterface.VariableIndex(1)

julia> MOI.set(model, MOI.VariableName(), x, "x_var")

julia> MOI.add_constraint(model, x, MOI.ZeroOne())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(1)

julia> MOI.set(model, MOI.ObjectiveFunction{typeof(x)}(), x)

julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)

julia> print(model)
Maximize VariableIndex:
 x_var

Subject to:

VariableIndex-in-ZeroOne
 x_var ∈ {0, 1}
```

Use `Utilities.latex_formulation` to display the model in LaTeX form:

```
julia> MOI.Utilities.latex_formulation(model)
$$ \begin{aligned}
\max\quad & x\_var \\
\text{Subject to}\\
 & \text{VariableIndex-in-ZeroOne} \\
 & x\_var \in \{0, 1\} \\
\end{aligned} $$
```

Tip

In IJulia, calling `print` or ending a cell with `Utilities.latex_formulation` will render the model in LaTeX.

Utilities.MatrixOfConstraints

The constraints of `Utilities.Model` are stored as a vector of tuples of function and set in a `Utilities.VectorOfConstraints`. Other representations can be used by parametrizing the type `Utilities.GenericModel` (resp. `Utilities.GenericOptimizer`). For instance, if all non-`VariableIndex` constraints are affine, the coefficients of all the constraints can be stored in a single sparse matrix using `Utilities.MatrixOfConstraints`. The constraints storage can even be customized up to a point where it exactly matches the storage of the solver of interest, in which case `copy_to` can be implemented for the solver by calling `copy_to` to this custom model.

For instance, Clp defines the following model

```
MOI.Utilities.@product_of_scalar_sets(LP, MOI.EqualTo{T}, MOI.LessThan{T}, MOI.GreaterThan{T})
const Model = MOI.Utilities.GenericModel{
    Float64,
    MOI.Utilities.MatrixOfConstraints{
        Float64,
        MOI.Utilities.MutableSparseMatrixCSC{Float64,Cint,MOI.Utilities.ZeroBasedIndexing},
        MOI.Utilities.Hyperrectangle{Float64},
        LP{Float64},
    },
}
```

The `copy_to` operation can now be implemented as follows (assuming that the `Model` definition above is in the `Clp` module so that it can be referred to as `Model`, to be distinguished with `Utilities.Model`):

```
function _copy_to(dest::Optimizer, src::Model)
    @assert MOI.is_empty(dest)
    A = src.constraints.coefficients
    row_bounds = src.constraints.constants
    Clp_loadProblem(
        dest,
        A.n,
        A.m,
        A.colptr,
        A.rowval,
        A.nzval,
        src.lower_bound,
        src.upper_bound,
        # (...) objective vector (omitted),
        row_bounds.lower,
        row_bounds.upper,
    )
    # Set objective sense and constant (omitted)
    return
end

function MOI.copy_to(dest::Optimizer, src::Model)
    _copy_to(dest, src)
    return MOI.Utilities.identity_index_map(src)
end

function MOI.copy_to(
```

```
    dest::Optimizer,
    src::MOI.Utilities.UniversalFallback{Model},
)
    # Copy attributes from `src` to `dest` and error in case any unsupported
    # constraints or attributes are set in `UniversalFallback`.
    return MOI.copy_to(dest, src.model)
end

function MOI.copy_to(
    dest::Optimizer,
    src::MOI.ModelLike,
)
    model = Model()
    index_map = MOI.copy_to(model, src)
    _copy_to(dest, model)
    return index_map
end
```

ModelFilter

Utilities provides `Utilities.ModelFilter` as a useful tool to copy a subset of a model. For example, given an infeasible model, we can copy the irreducible infeasible subsystem (for models implementing `ConstraintConflictStatus`) as follows:

```
my_filter(::Any) = true
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
index_map = MOI.copy_to(dest, filtered_src)
```

Fallbacks

The value of some attributes can be inferred from the value of other attributes.

For example, the value of `ObjectiveValue` can be computed using `ObjectiveFunction` and `VariablePrimal`.

When a solver gives direct access to an attribute, it is better to return this value. However, if this is not the case, `Utilities.get_fallback` can be used instead. For example:

```
function MOI.get(model::Optimizer, attr::MOI.ObjectiveFunction)
    return MOI.Utilities.get_fallback(model, attr)
end
```

DoubleDicts

When writing MOI interfaces, we often need to handle situations in which we map `ConstraintIndex`s to different values. For example, to a string for `ConstraintName`.

One option is to use a dictionary like `Dict{MOI.ConstraintIndex,String}`. However, this incurs a performance cost because the key is not a concrete type.

The DoubleDicts submodule helps this situation by providing two types main types `Utilities.DoubleDicts.DoubleDict` and `Utilities.DoubleDicts.IndexDoubleDict`. These types act like normal dictionaries, but internally they use more efficient dictionaries specialized to the type of the function-set pair.

The most common usage of a `DoubleDict` is in the `index_map` returned by `copy_to`. Performance can be improved, by using a function barrier. That is, instead of code like:

```
index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
end
```

use instead:

```
function function_barrier(
    dest,
    src,
    index_map::MOI.Utilities.DoubleDicts.IndexDoubleDictInner{F,S},
) where {F,S}
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
    return
end

index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
    function_barrier(dest, src, index_map[F, S])
end
```

29.2 API Reference

Utilities.Model

`MathOptInterface.Utilities.Model` – Type.

An implementation of `ModelLike` that supports all functions and sets defined in MOI. It is parameterized by the coefficient type.

Examples

```
model = Model{Float64}()
x = add_variable(model)
```

`source`

Utilities.UniversalFallback

`MathOptInterface.Utilities.UniversalFallback` – Type.

```
UniversalFallback
```

The `UniversalFallback` can be applied on a `MathOptInterface.ModelLike` model to create the model `UniversalFallback(model)` supporting any constraint and attribute. This allows to have a specialized implementation in `model` for performance critical constraints and attributes while still supporting other attributes with a small performance penalty. Note that `model` is unaware of constraints and attributes stored by `UniversalFallback` so this is not appropriate if `model` is an optimizer (for this reason, `MathOptInterface.optimize!` has not been implemented). In that case, optimizer bridges should be used instead.

source

Utilities.@model

`MathOptInterface.Utilities.@model` – Macro.

```
macro model(
    model_name,
    scalar_sets,
    typed_scalar_sets,
    vector_sets,
    typed_vector_sets,
    scalar_functions,
    typed_scalar_functions,
    vector_functions,
    typed_vector_functions,
    is_optimizer = false
)
```

Creates a type `model_name` implementing the MOI model interface and containing `scalar_sets` scalar sets `typed_scalar_sets` typed scalar sets, `vector_sets` vector sets, `typed_vector_sets` typed vector sets, `scalar_functions` scalar functions, `typed_scalar_functions` typed scalar functions, `vector_functions` vector functions and `typed_vector_functions` typed vector functions. To give no set/function, write `()`, to give one set S, write `(S,)`.

The function `MathOptInterface.VariableIndex` should not be given in `scalar_functions`. The model supports `MathOptInterface.VariableIndex`-in-S constraints where S is `MathOptInterface.EqualTo`, `MathOptInterface.Gr` `MathOptInterface.LessThan`, `MathOptInterface.Interval`, `MathOptInterface.Integer`, `MathOptInterface.ZeroOne`, `MathOptInterface.Semicontinuous` or `MathOptInterface.Semiinteger`. The sets supported with the `MathOptInterface.VariableIndex` cannot be controlled from the macro, use the `UniversalFallback` to support more sets.

This macro creates a model specialized for specific types of constraint, by defining specialized structures and methods. To create a model that, in addition to be optimized for specific constraints, also support arbitrary constraints and attributes, use `UniversalFallback`.

If `is_optimizer = true`, the resulting struct is a of `GenericOptimizer`, which is a subtype of `MathOptInterface.AbstractOp` otherwise, it is a `GenericModel`, which is a subtype of `MathOptInterface.ModelLike`.

Examples

The model describing an linear program would be:

```
@model(LPModel,                                                   # Name of model
      (),                                                         # untyped scalar sets
      (MOI.EqualTo, MOI.GreaterThan, MOI.LessThan, MOI.Interval), #   typed scalar sets
      (MOI.Zeros, MOI.Nonnegatives, MOI.Nonpositives),            # untyped vector sets
      (),                                                         #   typed vector sets
      (),                                                         # untyped scalar functions
      (MOI.ScalarAffineFunction,),                                #   typed scalar functions
      (MOI.VectorOfVariables,),                                   # untyped vector functions
      (MOI.VectorAffineFunction,),                                #   typed vector functions
```

```
    false
)
```

Let MOI denote MathOptInterface, MOIU denote MOI.Utilities. The macro would create the following types with struct_of_constraint_code:

```
struct LPModelScalarConstraints{T, C1, C2, C3, C4} <: MOIU.StructOfConstraints
    moi_equalto::C1
    moi_greaterthan::C2
    moi_lessthan::C3
    moi_interval::C4
end
struct LPModelVectorConstraints{T, C1, C2, C3} <: MOIU.StructOfConstraints
    moi_zeros::C1
    moi_nonnegatives::C2
    moi_nonpositives::C3
end
struct LPModelFunctionConstraints{T} <: MOIU.StructOfConstraints
    moi_scalaraffinefunction::LPModelScalarConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.EqualTo{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.GreaterThan{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.LessThan{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.Interval{T}}
    }
    moi_vectorofvariables::LPModelVectorConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Zeros},
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Nonnegatives},
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Nonpositives}
    }
    moi_vectoraffinefunction::LPModelVectorConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Zeros},
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Nonnegatives},
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Nonpositives}
    }
end
const LPModel{T} =
↪ MOIU.GenericModel{T,MOIU.ObjectiveContainer{T},MOIU.VariablesContainer{T},LPModelFunctionConstraints{T}}
```

The type LPModel implements the MathOptInterface API except methods specific to optimizers like optimize! or get with VariablePrimal.

source

MathOptInterface.Utilities.GenericModel – Type.

```
mutable struct GenericModel{T,O,V,C} <: AbstractModelLike{T}
```

Implements a model supporting coefficients of type T and:

- An objective function stored in .objective::O
- Variables and VariableIndex constraints stored in .variable_bounds::V
- F-in-S constraints (excluding VariableIndex constraints) stored in .constraints::C

All interactions should take place via the MOI interface, so the types O, V, and C should implement the API as needed for their functionality.

source

MathOptInterface.Utilities.GenericOptimizer – Type.

```
mutable struct GenericOptimizer{T,O,V,C} <: AbstractOptimizer{T}
```

Implements a model supporting coefficients of type T and:

- An objective function stored in `.objective::O`
- Variables and `VariableIndex` constraints stored in `.variable_bounds::V`
- F-in-S constraints (excluding `VariableIndex` constraints) stored in `.constraints::C`

All interactions should take place via the MOI interface, so the types O, V, and C should implement the API as needed for their functionality.

source

.objective MathOptInterface.Utilities.ObjectiveContainer – Type.

```
ObjectiveContainer{T}
```

A helper struct to simplify the handling of objective functions in Utilities.Model.

source

.variables MathOptInterface.Utilities.VariablesContainer – Type.

```
struct VariablesContainer{T} <: AbstractVectorBounds
    set_mask::Vector{UInt16}
    lower::Vector{T}
    upper::Vector{T}
end
```

A struct for storing variables and VariableIndex-related constraints. Used in `MOI.Utilities.Model` by default.

source

.constraints MathOptInterface.Utilities.VectorOfConstraints – Type.

```
mutable struct VectorOfConstraints{
    F<:MOI.AbstractFunction,
    S<:MOI.AbstractSet,
} <: MOI.ModelLike
    constraints::CleverDicts.CleverDict{
        MOI.ConstraintIndex{F,S},
        Tuple{F,S},
        typeof(CleverDicts.key_to_index),
        typeof(CleverDicts.index_to_key),
    }
end
```

A struct storing F-in-S constraints as a mapping between the constraint indices to the corresponding tuple of function and set.

source

MathOptInterface.Utilities.StructOfConstraints – Type.

```
abstract type StructOfConstraints <: MOI.ModelLike end
```

A struct storing a subfields other structs storing constraints of different types.

See Utilities.@struct_of_constraints_by_function_types and Utilities.@struct_of_constraints_by_set_types.

source

MathOptInterface.Utilities.@struct_of_constraints_by_function_types – Macro.

```
Utilities.@struct_of_constraints_by_function_types(name, func_types...)
```

Given a vector of n function types (F1, F2,..., Fn) in func_types, defines a subtype of StructOfConstraints of name name and which type parameters {T, C1, C2, ..., Cn}. It contains n field where the ith field has type Ci and stores the constraints of function type Fi.

The expression Fi can also be a union in which case any constraint for which the function type is in the union is stored in the field with type Ci.

source

MathOptInterface.Utilities.@struct_of_constraints_by_set_types – Macro.

```
Utilities.@struct_of_constraints_by_set_types(name, func_types...)
```

Given a vector of n set types (S1, S2,..., Sn) in func_types, defines a subtype of StructOfConstraints of name name and which type parameters {T, C1, C2, ..., Cn}. It contains n field where the ith field has type Ci and stores the constraints of set type Si. The expression Si can also be a union in which case any constraint for which the set type is in the union is stored in the field with type Ci. This can be useful if Ci is a MatrixOfConstraints in order to concatenate the coefficients of constraints of several different set types in the same matrix.

source

MathOptInterface.Utilities.struct_of_constraint_code – Function.

```
struct_of_constraint_code(struct_name, types, field_types = nothing)
```

Given a vector of n Union{SymbolFun,_UnionSymbolFS{SymbolFun}} or Union{SymbolSet,_UnionSymbolFS{SymbolSet}} in types, defines a subtype of StructOfConstraints of name name and which type parameters {T, F1, F2, ..., Fn} if field_types is nothing and a {T} otherwise. It contains n field where the ith field has type Ci if field_types is nothing and type field_types[i] otherwise. If types is vector of Union{SymbolFun,_UnionSymbolFS{SymbolFun}} (resp. Union{SymbolSet,_UnionSymbolFS{SymbolSet}}) then the constraints of that function (resp. set) type are stored in the corresponding field.

This function is used by the macros @model, @struct_of_constraints_by_function_types and @struct_of_constraints_b

source

Caching optimizer

MathOptInterface.Utilities.CachingOptimizer – Type.

```
CachingOptimizer
```

CachingOptimizer is an intermediate layer that stores a cache of the model and links it with an optimizer. It supports incremental model construction and modification even when the optimizer doesn't.

A CachingOptimizer may be in one of three possible states (CachingOptimizerState):

- NO_OPTIMIZER: The CachingOptimizer does not have any optimizer.
- EMPTY_OPTIMIZER: The CachingOptimizer has an empty optimizer. The optimizer is not synchronized with the cached model.
- ATTACHED_OPTIMIZER: The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

A CachingOptimizer has two modes of operation (CachingOptimizerMode):

- MANUAL: The only methods that change the state of the CachingOptimizer are Utilities.reset_optimizer, Utilities.drop_optimizer, and Utilities.attach_optimizer. Attempting to perform an operation in the incorrect state results in an error.
- AUTOMATIC: The CachingOptimizer changes its state when necessary. For example, optimize! will automatically call attach_optimizer (an optimizer must have been previously set). Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to EMPTY_OPTIMIZER mode.

source

MathOptInterface.Utilities.attach_optimizer – Function.

```
attach_optimizer(model::CachingOptimizer)
```

Attaches the optimizer to model, copying all model data into it. Can be called only from the EMPTY_OPTIMIZER state. If the copy succeeds, the CachingOptimizer will be in state ATTACHED_OPTIMIZER after the call, otherwise an error is thrown; see MathOptInterface.copy_to for more details on which errors can be thrown.

source

MathOptInterface.Utilities.reset_optimizer – Function.

```
reset_optimizer(m::CachingOptimizer, optimizer::MOI.AbstractOptimizer)
```

Sets or resets m to have the given empty optimizer optimizer.

Can be called from any state. An assertion error will be thrown if optimizer is not empty.

The CachingOptimizer m will be in state EMPTY_OPTIMIZER after the call.

source

```
reset_optimizer(m::CachingOptimizer)
```

Detaches and empties the current optimizer. Can be called from ATTACHED_OPTIMIZER or EMPTY_OPTIMIZER state. The CachingOptimizer will be in state EMPTY_OPTIMIZER after the call.

source

MathOptInterface.Utilities.drop_optimizer – Function.

```
drop_optimizer(m::CachingOptimizer)
```

Drops the optimizer, if one is present. Can be called from any state. The CachingOptimizer will be in state NO_OPTIMIZER after the call.

source

MathOptInterface.Utilities.state – Function.

```
state(m::CachingOptimizer)::CachingOptimizerState
```

Returns the state of the CachingOptimizer m. See Utilities.CachingOptimizer.

source

MathOptInterface.Utilities.mode – Function.

```
mode(m::CachingOptimizer)::CachingOptimizerMode
```

Returns the operating mode of the CachingOptimizer m. See Utilities.CachingOptimizer.

source

Mock optimizer

MathOptInterface.Utilities.MockOptimizer – Type.

```
MockOptimizer
```

MockOptimizer is a fake optimizer especially useful for testing. Its main feature is that it can store the values that should be returned for each attribute.

source

Printing

MathOptInterface.Utilities.latex_formulation – Function.

```
latex_formulation(model::MOI.ModelLike; kwargs...)
```

Wrap model in a type so that it can be pretty-printed as text/latex in a notebook like IJulia, or in Documenter.

To render the model, end the cell with latex_formulation(model), or call display(latex_formulation(model)) in to force the display of the model from inside a function.

Possible keyword arguments are:

- simplify_coefficients : Simplify coefficients if possible by omitting them or removing trailing zeros.
- default_name : The name given to variables with an empty name.
- print_types : Print the MOI type of each function and set for clarity.

source

Copy utilities

MathOptInterface.Utilities.default_copy_to – Function.

```
default_copy_to(dest::MOI.ModelLike, src::MOI.ModelLike)
```

A default implementation of MOI.copy_to(dest, src) for models that implement the incremental interface, i.e., MOI.supports_incremental_interface returns true.

source

MathOptInterface.Utilities.IndexMap – Type.

```
IndexMap()
```

The dictionary-like object returned by MathOptInterface.copy_to.

source

MathOptInterface.Utilities.identity_index_map – Function.

```
identity_index_map(model::MOI.ModelLike)
```

Return an IndexMap that maps all variable and constraint indices of model to themselves.

source

MathOptInterface.Utilities.ModelFilter – Type.

```
ModelFilter(filter::Function, model::MOI.ModelLike)
```

A layer to filter out various components of model.

The filter function takes a single argument, which is eacy element from the list returned by the attributes below. It returns true if the element should be visible in the filtered model and false otherwise.

The components that are filtered are:

- Entire constraint types via:
  - MOI.ListOfConstraintTypesPresent
- Individual constraints via:
  - MOI.ListOfConstraintIndices{F,S}
- Specific attributes via:
  - MOI.ListOfModelAttributesSet
  - MOI.ListOfConstraintAttributesSet
  - MOI.ListOfVariableAttributesSet

Warning
The list of attributes filtered may change in a future release. You should write functions that are generic and not limited to the five types listed above. Thus, you should probably define a fallback filter(::Any) = true.

See below for examples of how this works.

Note
This layer has a limited scope. It is intended by be used in conjunction with MOI.copy_to.

Example: copy model excluding integer constraints

Use the do syntax to provide a single function.

```
filtered_src = MOI.Utilities.ModelFilter(src) do item
    return item != (MOI.VariableIndex, MOI.Integer)
end
MOI.copy_to(dest, filtered_src)
```

Example: copy model excluding names

Use type dispatch to simplify the implementation:

```
my_filter(::Any) = true  # Note the generic fallback!
my_filter(::MOI.VariableName) = false
my_filter(::MOI.ConstraintName) = false
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

Example: copy irreducible infeasible subsystem

```
my_filter(::Any) = true  # Note the generic fallback!
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

source

MatrixOfConstraints

`MathOptInterface.Utilities.MatrixOfConstraints` – Type.

```
mutable struct MatrixOfConstraints{T,AT,BT,ST} <: MOI.ModelLike
    coefficients::AT
    constants::BT
    sets::ST
    caches::Vector{Any}
    are_indices_mapped::Vector{BitSet}
    final_touch::Bool
end
```

Represent `ScalarAffineFunction` and `VectorAffinefunction` constraints in a matrix form where the linear coefficients of the functions are stored in the `coefficients` field, the constants of the functions or sets are stored in the `constants` field. Additional information about the sets are stored in the `sets` field.

This model can only be used as the `constraints` field of a `MOI.Utilities.AbstractModel`.

When the constraints are added, they are stored in the caches field. They are only loaded in the `coefficients` and `constants` fields once `MOI.Utilities.final_touch` is called. For this reason, `MatrixOfConstraints` should not be used by an incremental interface. Use `MOI.copy_to` instead.

The constraints can be added in two different ways:

1. With `add_constraint`, in which case a canonicalized copy of the function is stored in `caches`.

2. With `pass_nonvariable_constraints`, in which case the functions and sets are stored themselves in `caches` without mapping the variable indices. The corresponding index in `caches` is added in `are_indices_mapped`. This avoids doing a copy of the function in case the getter of `CanonicalConstraintFunction` does not make a copy for the source model, e.g., this is the case of `VectorOfConstraints`.

We illustrate this with an example. Suppose a model is copied from a `src::MOI.Utilities.Model` to a bridged model with a `MatrixOfConstraints`. For all the types that are not bridged, the constraints will be copied with `pass_nonvariable_constraints`. Hence the functions stored in `caches` are exactly the same as the ones stored in `src`. This is ok since this is only during the `copy_to` operation during which `src` cannot be modified. On the other hand, for the types that are bridged, the functions added may contain duplicates even if the functions did not contain duplicates in `src` so duplicates are removed with `MOI.Utilities.canonical`.

Interface

The `.coefficients::AT` type must implement:

- `AT()`
- `MOI.empty(::AT)!`
- `MOI.Utilities.add_column`
- `MOI.Utilities.set_number_of_rows`
- `MOI.Utilities.allocate_terms`
- `MOI.Utilities.load_terms`
- `MOI.Utilities.final_touch`

The `.constants::BT` type must implement:

- `BT()`
- `Base.empty!(::BT)`
- `Base.resize(::BT)`
- `MOI.Utilities.load_constants`
- `MOI.Utilities.function_constants`
- `MOI.Utilities.set_from_constants`

The `.sets::ST` type must implement:

- `ST()`
- `MOI.is_empty(::ST)`
- `MOI.empty(::ST)`
- `MOI.dimension(::ST)`
- `MOI.is_valid(::ST, ::MOI.ConstraintIndex)`
- `MOI.get(::ST, ::MOI.ListOfConstraintTypesPresent)`
- `MOI.get(::ST, ::MOI.NumberOfConstraints)`
- `MOI.get(::ST, ::MOI.ListOfConstraintIndices)`
- `MOI.Utilities.set_types`
- `MOI.Utilities.set_index`
- `MOI.Utilities.add_set`
- `MOI.Utilities.rows`
- `MOI.Utilities.final_touch`

source

.coefficients MathOptInterface.Utilities.add_column – Function.

```
add_column(coefficients)::Nothing
```

Tell `coefficients` to pre-allocate datastructures as needed to store one column.

source

MathOptInterface.Utilities.allocate_terms – Function.

```
allocate_terms(coefficients, index_map, func)::Nothing
```

Tell `coefficients` that the terms of the function `func` where the variable indices are mapped with `index_map` will be loaded with `load_terms`.

The function `func` must be canonicalized before calling `allocate_terms`. See `is_canonical`.

source

MathOptInterface.Utilities.set_number_of_rows – Function.

```
set_number_of_rows(coefficients, n)::Nothing
```

Tell `coefficients` to pre-allocate datastructures as needed to store `n` rows.

source

MathOptInterface.Utilities.load_terms – Function.

```
load_terms(coefficients, index_map, func, offset)::Nothing
```

Loads the terms of `func` to `coefficients`, mapping the variable indices with `index_map`.

The `i`th dimension of `func` is loaded at the `(offset + i)`th row of `coefficients`.

The function must be allocated first with `allocate_terms`.

The function `func` must be canonicalized, see `is_canonical`.

source

MathOptInterface.Utilities.final_touch – Function.

```
final_touch(coefficients)::Nothing
```

Informs the `coefficients` that all functions have been added with `load_terms`. No more modification is allowed unless `MOI.empty!` is called.

```
final_touch(sets)::Nothing
```

Informs the `sets` that all functions have been added with `add_set`. No more modification is allowed unless `MOI.empty!` is called.

source

MathOptInterface.Utilities.extract_function – Function.

```
extract_function(coefficients, row::Integer, constant::T) where {T}
```

Return the `MOI.ScalarAffineFunction{T}` function corresponding to row `row` in `coefficients`.

```
extract_function(
    coefficients,
    rows::UnitRange,
    constants::Vector{T},
) where{T}
```

Return the `MOI.VectorAffineFunction{T}` function corresponding to rows `rows` in `coefficients`.

source

`MathOptInterface.Utilities.MutableSparseMatrixCSC` – Type.

```
mutable struct MutableSparseMatrixCSC{Tv,Ti<:Integer,I<:AbstractIndexing}
    indexing::I
    m::Int
    n::Int
    colptr::Vector{Ti}
    rowval::Vector{Ti}
    nzval::Vector{Tv}
    nz_added::Vector{Ti}
end
```

Matrix type loading sparse matrices in the Compressed Sparse Column format. The indexing used is `indexing`, see `AbstractIndexing`. The other fields have the same meaning than for `SparseArrays.SparseMatrixCSC` except that the indexing is different unless `indexing` is `OneBasedIndexing`. In addition, `nz_added` is used to cache the number of non-zero terms that have been added to each column due to the incremental nature of `load_terms`.

The matrix is loaded in 5 steps:

1. `MOI.empty!` is called.
2. `MOI.Utilities.add_column` and `MOI.Utilities.allocate_terms` are called in any order.
3. `MOI.Utilities.set_number_of_rows` is called.
4. `MOI.Utilities.load_terms` is called for each affine function.
5. `MOI.Utilities.final_touch` is called.

source

`MathOptInterface.Utilities.AbstractIndexing` – Type.

```
abstract type AbstractIndexing end
```

Indexing to be used for storing the row and column indices of `MutableSparseMatrixCSC`. See `ZeroBasedIndexing` and `OneBasedIndexing`.

source

`MathOptInterface.Utilities.ZeroBasedIndexing` – Type.

```
struct ZeroBasedIndexing <: AbstractIndexing end
```

Zero-based indexing: the `i`th row or column has index `i - 1`. This is useful when the vectors of row and column indices need to be communicated to a library using zero-based indexing such as C libraries.

source

MathOptInterface.Utilities.OneBasedIndexing – Type.

```
struct ZeroBasedIndexing <: AbstractIndexing end
```

One-based indexing: the ith row or column has index i. This enables an allocation-free conversion of MutableSparseMatrixCSC to SparseArrays.SparseMatrixCSC.

source

.constants MathOptInterface.Utilities.load_constants – Function.

```
load_constants(constants, offset, func_or_set)::Nothing
```

This function loads the constants of func_or_set in constants at an offset of offset. Where offset is the sum of the dimensions of the constraints already loaded. The storage should be preallocated with resize! before calling this function.

This function should be implemented to be usable as storage of constants for MatrixOfConstraints.

The constants are loaded in three steps:

1. Base.empty! is called.
2. Base.resize! is called with the sum of the dimensions of all constraints.
3. MOI.Utilities.load_constants is called for each function for vector constraint or set for scalar constraint.

source

MathOptInterface.Utilities.function_constants – Function.

```
function_constants(constants, rows)
```

This function returns the function constants that were loaded with load_constants at the rows rows.

This function should be implemented to be usable as storage of constants for MatrixOfConstraints.

source

MathOptInterface.Utilities.set_from_constants – Function.

```
set_from_constants(constants, S::Type, rows)::S
```

This function returns an instance of the set S for which the constants where loaded with load_constants at the rows rows.

This function should be implemented to be usable as storage of constants for MatrixOfConstraints.

source

MathOptInterface.Utilities.Hyperrectangle – Type.

```
struct Hyperrectangle{T} <: AbstractVectorBounds
    lower::Vector{T}
    upper::Vector{T}
end
```

A struct for the .constants field in MatrixOfConstraints.

source

.sets MathOptInterface.Utilities.set_index – Function.

```
set_index(sets, ::Type{S})::Union{Int,Nothing} where {S<:MOI.AbstractSet}
```

Return an integer corresponding to the index of the set type in the list given by set_types.

If S is not part of the list, return nothing.

source

MathOptInterface.Utilities.set_types – Function.

```
set_types(sets)::Vector{Type}
```

Return the list of the types of the sets allowed in sets.

source

MathOptInterface.Utilities.add_set – Function.

```
add_set(sets, i)::Int64
```

Add a scalar set of type index i.

```
add_set(sets, i, dim)::Int64
```

Add a vector set of type index i and dimension dim.

Both methods return a unique Int64 of the set that can be used to reference this set.

source

MathOptInterface.Utilities.rows – Function.

```
rows(sets, ci::MOI.ConstraintIndex)::Union{Int,UnitRange{Int}}
```

Return the rows in 1:MOI.dimension(sets) corresponding to the set of id ci.value.

For scalar sets, this returns an Int. For vector sets, this returns an UnitRange{Int}.

source

MathOptInterface.Utilities.num_rows – Function.

```
num_rows(sets::OrderedProductOfSets, ::Type{S}) where {S}
```

Return the number of rows corresponding to a set of type S. That is, it is the sum of the dimensions of the sets of type S.

source

MathOptInterface.Utilities.set_with_dimension – Function.

```
set_with_dimension(::Type{S}, dim) where {S<:MOI.AbstractVectorSet}
```

Returns the instance of S of MathOptInterface.dimension dim. This needs to be implemented for sets of type S to be useable with MatrixOfConstraints.

source

MathOptInterface.Utilities.ProductOfSets – Type.

```
abstract type ProductOfSets{T} end
```

Represents a cartesian product of sets of given types.

source

MathOptInterface.Utilities.MixOfScalarSets – Type.

```
abstract type MixOfScalarSets{T} <: ProductOfSets{T} end
```

Product of scalar sets in the order the constraints are added, mixing the constraints of different types.

Use @mix_of_scalar_sets to generate a new subtype.

source

MathOptInterface.Utilities.@mix_of_scalar_sets – Macro.

```
@mix_of_scalar_sets(name, set_types...)
```

Generate a new MixOfScalarSets subtype.

Example

```
@mix_of_scalar_sets(
    MixedIntegerLinearProgramSets,
    MOI.GreaterThan{T},
    MOI.LessThan{T},
    MOI.EqualTo{T},
    MOI.Integer,
)
```

source

MathOptInterface.Utilities.OrderedProductOfSets – Type.

```
abstract type OrderedProductOfSets{T} <: ProductOfSets{T} end
```

Product of sets in the order the constraints are added, grouping the constraints of the same types contiguously.

Use @product_of_sets to generate new subtypes.

source

MathOptInterface.Utilities.@product_of_sets – Macro.

```
@product_of_sets(name, set_types...)
```

Generate a new OrderedProductOfSets subtype.

Example

```
@product_of_sets(
    LinearOrthants,
    MOI.Zeros,
    MOI.Nonnegatives,
    MOI.Nonpositives,
    MOI.ZeroOne,
)
```

source

Fallbacks

MathOptInterface.Utilities.get_fallback – Function.

```
get_fallback(model::MOI.ModelLike, ::MOI.ObjectiveValue)
```

Compute the objective function value using the VariablePrimal results and the ObjectiveFunction value.

source

```
get_fallback(model::MOI.ModelLike, ::MOI.DualObjectiveValue, T::Type)::T
```

Compute the dual objective value of type T using the ConstraintDual results and the ConstraintFunction and ConstraintSet values. Note that the nonlinear part of the model is ignored.

source

```
get_fallback(model::MOI.ModelLike, ::MOI.ConstraintPrimal,
             constraint_index::MOI.ConstraintIndex)
```

Compute the value of the function of the constraint of index constraint_index using the VariablePrimal results and the ConstraintFunction values.

source

```
get_fallback(model::MOI.ModelLike, attr::MOI.ConstraintDual,
             ci::MOI.ConstraintIndex{Union{MOI.VariableIndex,
                                            MOI.VectorOfVariables}})
```

Compute the dual of the constraint of index ci using the ConstraintDual of other constraints and the ConstraintFunction values. Throws an error if some constraints are quadratic or if there is one another MOI.VariableIndex-in-S or MOI.VectorOfVariables-in-S constraint with one of the variables in the function of the constraint ci.

source

Function utilities

The following utilities are available for functions:

MathOptInterface.Utilities.eval_variables – Function.

```
eval_variables(varval::Function, f::AbstractFunction)
```

Returns the value of function f if each variable index vi is evaluated as varval(vi). Note that varval should return a number, see substitute_variables for a similar function where varval returns a function.

source

MathOptInterface.Utilities.map_indices – Function.

```
map_indices(index_map::Function, attr::MOI.AnyAttribute, x::X)::X where {X}
```

Substitute any MOI.VariableIndex (resp. MOI.ConstraintIndex) in x by the MOI.VariableIndex (resp. MOI.ConstraintIndex) of the same type given by index_map(x).

When to implement this method for new types X

This function is used by implementations of MOI.copy_to on constraint functions, attribute values and submittable values. If you define a new attribute whose values x::X contain variable or constraint indices, you must also implement this function.

source

```
map_indices(
    variable_map::AbstractDict{T,T},
    x::X,
)::X where {T<:MOI.Index,X}
```

Shortcut for map_indices(vi -> variable_map[vi], x).

source

MathOptInterface.Utilities.substitute_variables – Function.

```
substitute_variables(variable_map::Function, x)
```

Substitute any MOI.VariableIndex in x by variable_map(x). The variable_map function returns either MOI.VariableIndex or MOI.ScalarAffineFunction, see eval_variables for a similar function where variable_map returns a number.

This function is used by bridge optimizers on constraint functions, attribute values and submittable values when at least one variable bridge is used hence it needs to be implemented for custom types that are meant to be used as attribute or submittable value.

WARNING: Don't use substitude_variables(::Function, ...) because Julia will not specialize on this. Use instead substitude_variables(::F, ...) where {F<:Function}.

source

MathOptInterface.Utilities.filter_variables – Function.

```
filter_variables(keep::Function, f::AbstractFunction)
```

Return a new function f with the variable vi such that !keep(vi) removed.

WARNING: Don't define filter_variables(::Function, ...) because Julia will not specialize on this. Define instead filter_variables(::F, ...) where {F<:Function}.

source

MathOptInterface.Utilities.remove_variable – Function.

```
remove_variable(f::AbstractFunction, vi::VariableIndex)
```

Return a new function f with the variable vi removed.

source

```
remove_variable(f::MOI.AbstractFunction, s::MOI.AbstractSet, vi::MOI.VariableIndex)
```

Return a tuple (g, t) representing the constraint f-in-s with the variable vi removed. That is, the terms containing the variable vi in the function f are removed and the dimension of the set s is updated if needed (e.g. when f is a VectorOfVariables with vi being one of the variables).

source

MathOptInterface.Utilities.all_coefficients – Function.

```
all_coefficients(p::Function, f::MOI.AbstractFunction)
```

Determine whether predicate `p` returns `true` for all coefficients of `f`, returning `false` as soon as the first coefficient of `f` for which `p` returns `false` is encountered (short-circuiting). Similar to `all`.

source

`MathOptInterface.Utilities.unsafe_add` – Function.

```
unsafe_add(t1::MOI.ScalarAffineTerm, t2::MOI.ScalarAffineTerm)
```

Sums the coefficients of `t1` and `t2` and returns an output `MOI.ScalarAffineTerm`. It is unsafe because it uses the `variable` of `t1` as the `variable` of the output without checking that it is equal to that of `t2`.

source

```
unsafe_add(t1::MOI.ScalarQuadraticTerm, t2::MOI.ScalarQuadraticTerm)
```

Sums the coefficients of `t1` and `t2` and returns an output `MOI.ScalarQuadraticTerm`. It is unsafe because it uses the `variable`'s of `t1` as the `variable`'s of the output without checking that they are the same (up to permutation) to those of `t2`.

source

```
unsafe_add(t1::MOI.VectorAffineTerm, t2::MOI.VectorAffineTerm)
```

Sums the coefficients of `t1` and `t2` and returns an output `MOI.VectorAffineTerm`. It is unsafe because it uses the `output_index` and `variable` of `t1` as the `output_index` and `variable` of the output term without checking that they are equal to those of `t2`.

source

`MathOptInterface.Utilities.isapprox_zero` – Function.

```
isapprox_zero(f::MOI.AbstractFunction, tol)
```

Return a `Bool` indicating whether the function `f` is approximately zero using `tol` as a tolerance.

Important note

This function assumes that `f` does not contain any duplicate terms, you might want to first call `canonical` if that is not guaranteed. For instance, given

```
f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.([1, -1], [x, x]), 0)`.
```

then `isapprox_zero(f)` is `false` but `isapprox_zero(MOIU.canonical(f))` is `true`.

source

`MathOptInterface.Utilities.modify_function` – Function.

```
modify_function(f::AbstractFunction, change::AbstractFunctionModification)
```

Return a new function `f` modified according to `change`.

source

`MathOptInterface.Utilities.zero_with_output_dimension` – Function.

```
zero_with_output_dimension(::Type{T}, output_dimension::Integer) where {T}
```

Create an instance of type T with the output dimension `output_dimension`.

This is mostly useful in Bridges, when code needs to be agnostic to the type of vector-valued function that is passed in.

source

The following functions can be used to canonicalize a function:

`MathOptInterface.Utilities.is_canonical` – Function.

```
is_canonical(f::Union{ScalarAffineFunction, VectorAffineFunction})
```

Returns a Bool indicating whether the function is in canonical form. See `canonical`.

source

```
is_canonical(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})
```

Returns a Bool indicating whether the function is in canonical form. See `canonical`.

source

`MathOptInterface.Utilities.canonical` – Function.

```
canonical(
    f::Union{
        ScalarAffineFunction,
        VectorAffineFunction,
        ScalarQuadraticFunction,
        VectorQuadraticFunction,
    },
)
```

Returns the function in a canonical form, i.e.

- A term appear only once.
- The coefficients are nonzero.
- The terms appear in increasing order of variable where there the order of the variables is the order of their value.
- For a `AbstractVectorFunction`, the terms are sorted in ascending order of output index.

The output of `canonical` can be assumed to be a copy of `f`, even for `VectorOfVariables`.

Examples

If x (resp. y, z) is `VariableIndex(1)` (resp. 2, 3). The canonical representation of `ScalarAffineFunction([y, x, z, x, z], [2, 1, 3, -2, -3], 5)` is `ScalarAffineFunction([x, y], [-1, 2], 5)`.

source

`MathOptInterface.Utilities.canonicalize!` – Function.

```
canonicalize!(f::Union{ScalarAffineFunction, VectorAffineFunction})
```

Convert a function to canonical form in-place, without allocating a copy to hold the result. See `canonical`.

source

```
canonicalize!(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})
```

Convert a function to canonical form in-place, without allocating a copy to hold the result. See `canonical`.

source

The following functions can be used to manipulate functions with basic algebra:

`MathOptInterface.Utilities.scalar_type` – Function.

```
scalar_type(F::Type{<:MOI.AbstractVectorFunction})
```

Type of functions obtained by indexing objects obtained by calling `eachscalar` on functions of type `F`.

source

`MathOptInterface.Utilities.scalarize` – Function.

```
scalarize(func::MOI.VectorOfVariables, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.SingleVariable}`.

See also `eachscalar`.

source

```
scalarize(func::MOI.VectorAffineFunction{T}, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.ScalarAffineFunction{T}}`.

See also `eachscalar`.

source

```
scalarize(func::MOI.VectorQuadraticFunction{T}, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.ScalarQuadraticFunction{T}`

See also `eachscalar`.

source

`MathOptInterface.Utilities.eachscalar` – Function.

```
eachscalar(f::MOI.AbstractVectorFunction)
```

Returns an iterator for the scalar components of the vector function.

See also `scalarize`.

source

```
eachscalar(f::MOI.AbstractVector)
```

Returns an iterator for the scalar components of the vector.

source

`MathOptInterface.Utilities.promote_operation` – Function.

```
promote_operation(
    op::Function,
    ::Type{T},
    ArgsTypes::Type{<:Union{T, MOI.AbstractFunction}}...,
) where {T}
```

Returns the type of the `MOI.AbstractFunction` returned to the call `operate(op, T, args...)` where the types of the arguments `args` are `ArgsTypes`.

source

`MathOptInterface.Utilities.operate` – Function.

```
operate(
    op::Function,
    ::Type{T},
    args::Union{T,MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractFunction` representing the function resulting from the operation `op(args...)` on functions of coefficient type T. No argument can be modified.

source

`MathOptInterface.Utilities.operate!` – Function.

```
operate!(
    op::Function,
    ::Type{T},
    args::Union{T, MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractFunction` representing the function resulting from the operation `op(args...)` on functions of coefficient type T. The first argument can be modified. The return type is the same than the method `operate(op, T, args...)` without `!`.

source

`MathOptInterface.Utilities.operate_output_index!` – Function.

```
operate_output_index!(
    op::Function,
    ::Type{T},
    output_index::Integer,
    func::MOI.AbstractVectorFunction
    args::Union{T, MOI.AbstractScalarFunction}...
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractVectorFunction` where the function at `output_index` is the result of the operation op applied to the function at `output_index` of `func` and `args`. The functions at output index different to `output_index` are the same as the functions at the same output index in `func`. The first argument can be modified.

source

`MathOptInterface.Utilities.vectorize` – Function.

```
vectorize(x::AbstractVector{MOI.VariableIndex})
```

Returns the vector of scalar affine functions in the form of a `MOI.VectorAffineFunction{T}`.

source

```
vectorize(funcs::AbstractVector{MOI.ScalarAffineFunction{T}}) where T
```

Returns the vector of scalar affine functions in the form of a `MOI.VectorAffineFunction{T}`.

source

```
vectorize(funcs::AbstractVector{MOI.ScalarQuadraticFunction{T}}) where T
```

Returns the vector of scalar quadratic functions in the form of a `MOI.VectorQuadraticFunction{T}`.

source

Constraint utilities

The following utilities are available for moving the function constant to the set for scalar constraints:

`MathOptInterface.Utilities.shift_constant` – Function.

```
shift_constant(set::MOI.AbstractScalarSet, offset)
```

Returns a new scalar set `new_set` such that `func`-in-`set` is equivalent to `func + offset`-in-`new_set`.

Only define this function if it makes sense to!

Use `supports_shift_constant` to check if the set supports shifting:

```
if supports_shift_constant(typeof(old_set))
    new_set = shift_constant(old_set, offset)
    f.constant = 0
    add_constraint(model, f, new_set)
else
    add_constraint(model, f, old_set)
end
```

See also `supports_shift_constant`.

Examples

The call `shift_constant(MOI.Interval(-2, 3), 1)` is equal to `MOI.Interval(-1, 4)`.

source

`MathOptInterface.Utilities.supports_shift_constant` – Function.

```
supports_shift_constant(::Type{S}) where {S<:MOI.AbstractSet}
```

Return `true` if `shift_constant` is defined for set `S`.

See also `shift_constant`.

source

`MathOptInterface.Utilities.normalize_and_add_constraint` – Function.

```
normalize_and_add_constraint(
    model::MOI.ModelLike,
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Adds the scalar constraint obtained by moving the constant term in `func` to the set in `model`. If `allow_modify_function` is `true` then the function `func` can be modified.

source

MathOptInterface.Utilities.normalize_constant – Function.

```
normalize_constant(
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Return the `func`-in-`set` constraint in normalized form. That is, if `func` is `MOI.ScalarQuadraticFunction` or `MOI.ScalarAffineFunction`, the constant is moved to the set. If `allow_modify_function` is `true` then the function `func` can be modified.

source

The following utility identifies those constraints imposing bounds on a given variable, and returns those bound values:

MathOptInterface.Utilities.get_bounds – Function.

```
get_bounds(model::MOI.ModelLike, ::Type{T}, x::MOI.VariableIndex)
```

Return a tuple `(lb, ub)` of type `Tuple{T, T}`, where `lb` and `ub` are lower and upper bounds, respectively, imposed on `x` in `model`.

source

The following utilities are useful when working with symmetric matrix cones.

MathOptInterface.Utilities.is_diagonal_vectorized_index – Function.

```
is_diagonal_vectorized_index(index::Base.Integer)
```

Return whether `index` is the index of a diagonal element in a `MOI.AbstractSymmetricMatrixSetTriangle` set.

source

MathOptInterface.Utilities.side_dimension_for_vectorized_dimension – Function.

```
side_dimension_for_vectorized_dimension(n::Integer)
```

Return the dimension `d` such that `MOI.dimension(MOI.PositiveSemidefiniteConeTriangle(d))` is `n`.

source

DoubleDicts

MathOptInterface.Utilities.DoubleDicts.DoubleDict – Type.

```
DoubleDict{V}
```

An optimized dictionary to map `MOI.ConstraintIndex` to values of type `V`.

Works as a `AbstractDict{MOI.ConstraintIndex,V}` with minimal differences.

If `V` is also a `MOI.ConstraintIndex`, use `IndexDoubleDict`.

Note that `MOI.ConstraintIndex` is not a concrete type, opposed to `MOI.ConstraintIndex{MOI.VariableIndex, MOI.Integers}`, which is a concrete type.

When looping through multiple keys of the same Function-in-Set type, use

```
inner = dict[F, S]
```

to return a type-stable DoubleDictInner.

source

MathOptInterface.Utilities.DoubleDicts.DoubleDictInner – Type.

```
DoubleDictInner{F,S,V}
```

A type stable inner dictionary of DoubleDict.

source

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDict – Type.

```
IndexDoubleDict
```

A specialized version of [DoubleDict] in which the values are of type MOI.ConstraintIndex

When looping through multiple keys of the same Function-in-Set type, use

```
inner = dict[F, S]
```

to return a type-stable IndexDoubleDictInner.

source

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDictInner – Type.

```
IndexDoubleDictInner{F,S}
```

A type stable inner dictionary of IndexDoubleDict.

source

Chapter 30

Test

30.1 Overview

The Test submodule

The `Test` submodule provides tools to help solvers implement unit tests in order to ensure they implement the MathOptInterface API correctly, and to check for solver-correctness.

We use a centralized repository of tests, so that if we find a bug in one solver, instead of adding a test to that particular repository, we add it here so that all solvers can benefit.

How to test a solver

The skeleton below can be used for the wrapper test file of a solver named `FooBar`.

```
# ============================ /test/MOI_wrapper.jl ============================
module TestFooBar

import FooBar
using MathOptInterface
using Test

const MOI = MathOptInterface

const OPTIMIZER = MOI.instantiate(
    MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
)

const BRIDGED = MOI.instantiate(
    MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
    with_bridge_type = Float64,
)

# See the docstring of MOI.Test.Config for other arguments.
const CONFIG = MOI.Test.Config(
    # Modify tolerances as necessary.
    atol = 1e-6,
    rtol = 1e-6,
    # Use MOI.LOCALLY_SOLVED for local solvers.
    optimal_status = MOI.OPTIMAL,
    # Pass attributes or MOI functions to `exclude` to skip tests that
```

```julia
    # rely on this functionality.
    exclude = Any[MOI.VariableName, MOI.delete],
)

"""
    runtests()

This function runs all functions in the this Module starting with `test_`.
"""
function runtests()
    for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
end

"""
    test_runtests()

This function runs all the tests in MathOptInterface.Test.

Pass arguments to `exclude` to skip tests for functionality that is not
implemented or that your solver doesn't support.
"""
function test_runtests()
    MOI.Test.runtests(
        BRIDGED,
        CONFIG,
        exclude = [
            "test_attribute_NumberOfThreads",
            "test_quadratic_",
        ],
        # This argument is useful to prevent tests from failing on future
        # releases of MOI that add new tests. Don't let this number get too far
        # behind the current MOI release though! You should periodically check
        # for new tests in order to fix bugs and implement new features.
        exclude_tests_after = v"0.10.5",
    )
    return
end

"""
    test_SolverName()

You can also write new tests for solver-specific functionality. Write each new
test as a function with a name beginning with `test_`.
"""
function test_SolverName()
    @test MOI.get(FooBar.Optimizer(), MOI.SolverName()) == "FooBar"
    return
end

end # module TestFooBar
```

```julia
# This line at tne end of the file runs all the tests!
TestFooBar.runtests()
```

Then modify your `runtests.jl` file to include the `MOI_wrapper.jl` file:

```julia
# ============================ /test/runtests.jl ============================

using Test

@testset "MOI" begin
    include("test/MOI_wrapper.jl")
end
```

> **Info**
>
> The optimizer `BRIDGED` constructed with `instantiate` automatically bridges constraints that are not supported by `OPTIMIZER` using the bridges listed in Bridges. It is recommended for an implementation of MOI to only support constraints that are natively supported by the solver and let bridges transform the constraint to the appropriate form. For this reason it is expected that tests may not pass if `OPTIMIZER` is used instead of `BRIDGED`.

How to debug a failing test

When writing a solver, it's likely that you will initially fail many tests! Some failures will be bugs, but other failures you may choose to exclude.

There are two ways to exclude tests:

- Exclude tests whose names contain a string using:

  ```julia
  MOI.Test.runtests(
      model,
      config;
      exclude = String["test_to_exclude", "test_conic_"],
  )
  ```

  This will exclude tests whose name contains either of the two strings provided.

- Exclude tests which rely on specific functionality using:

  ```julia
  MOI.Test.Config(exclude = Any[MOI.VariableName, MOI.optimize!])
  ```

  This will exclude tests which use the `MOI.VariableName` attribute, or which call `MOI.optimize!`.

Each test that fails can be independently called as:

```julia
model = FooBar.Optimizer()
config = MOI.Test.Config()
MOI.empty!(model)
MOI.Test.test_category_name_that_failed(model, config)
```

You can look-up the source code of the test that failed by searching for it in the `src/Test/test_category.jl` file.

Tip

Each test function also has a docstring that explains what the test is for. Use `? MOI.Test.test_category_name_that_faile` from the REPL to read it.

How to add a test

To detect bugs in solvers, we add new tests to `MOI.Test`.

As an example, ECOS errored calling `optimize!` twice in a row. (See ECOS.jl PR #72.) We could add a test to ECOS.jl, but that would only stop us from re-introducing the bug to ECOS.jl in the future, but it would not catch other solvers in the ecosystem with the same bug! Instead, if we add a test to `MOI.Test`, then all solvers will also check that they handle a double optimize call!

For this test, we care about correctness, rather than performance. therefore, we don't expect solvers to efficiently decide that they have already solved the problem, only that calling `optimize!` twice doesn't throw an error or give the wrong answer.

Step 1

Install the `MathOptInterface` julia package in dev mode (ref):

```
julia> ]
(@v1.6) pkg> dev MathOptInterface
```

Step 2

From here on, proceed with making the following changes in the `~/.julia/dev/MathOptInterface` folder (or equivalent dev path on your machine).

Step 3

Since the double-optimize error involves solving an optimization problem, add a new test to src/Test/UnitTests/solve.jl:

```
"""
    test_unit_optimize!_twice(model::MOI.ModelLike, config::Config)

Test that calling `MOI.optimize!` twice does not error.

This problem was first detected in ECOS.jl PR#72:
https://github.com/jump-dev/ECOS.jl/pull/72
"""
function test_unit_optimize!_twice(
    model::MOI.ModelLike,
    config::Config{T},
) where {T}
    # Use the `@requires` macro to check conditions that the test function
    # requires in order to run. Models failing this `@requires` check will
    # silently skip the test.
    @requires MOI.supports_constraint(
        model,
        MOI.VariableIndex,
        MOI.GreaterThan{Float64},
    )
    @requires _supports(config, MOI.optimize!)
    # If needed, you can test that the model is empty at the start of the test.
```

```
    # You can assume that this will be the case for tests run via `runtests`.
    # User's calling tests individually need to call `MOI.empty!` themselves.
    @test MOI.is_empty(model)
    # Create a simple model. Try to make this as simple as possible so that the
    # majority of solvers can run the test.
    x = MOI.add_variable(model)
    MOI.add_constraint(model, x, MOI.GreaterThan(one(T)))
    MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
    MOI.set(
        model,
        MOI.ObjectiveFunction{MOI.VariableIndex}(),
        x,
    )
    # The main component of the test: does calling `optimize!` twice error?
    MOI.optimize!(model)
    MOI.optimize!(model)
    # Check we have a solution.
    @test MOI.get(model, MOI.TerminationStatus()) == MOI.OPTIMAL
    # There is a three-argument version of `Base.isapprox` for checking
    # approximate equality based on the tolerances defined in `config`:
    @test isapprox(MOI.get(model, MOI.VariablePrimal(), x), one(T), config)
    # For code-style, these tests should always `return` `nothing`.
    return
end
```

> **Info**
>
> Make sure the function is agnoistic to the number type T! Don't assume it is a `Float64` capable solver!

We also need to write a test for the test. Place this function immediately below the test you just wrote in the same file:

```
function setup_test(
    ::typeof(test_unit_optimize!_twice),
    model::MOI.Utilities.MockOptimizer,
    ::Config,
)
    MOI.Utilities.set_mock_optimize!(
        model,
        (mock::MOI.Utilities.MockOptimizer) -> MOIU.mock_optimize!(
            mock,
            MOI.OPTIMAL,
            (MOI.FEASIBLE_POINT, [1.0]),
        ),
    )
    return
end
```

Finally, you also need to implement `Test.version_added`. If we added this test when the latest released version of MOI was `v0.10.5`, define:

```
version_added(::typeof(test_unit_optimize!_twice)) = v"0.10.6"
```

Step 6

Commit the changes to git from `~/.julia/dev/MathOptInterface` and submit the PR for review.

> **Tip**
>
> If you need help writing a test, open an issue on GitHub, or ask the Developer Chatroom

30.2 API Reference

The Test submodule

Functions to help test implementations of MOI. See The Test submodule for more details.

`MathOptInterface.Test.Config` – Type.

```
Config(
    ::Type{T} = Float64;
    atol::Real = Base.rtoldefault(T),
    rtol::Real = Base.rtoldefault(T),
    optimal_status::MOI.TerminationStatusCode = MOI.OPTIMAL,
    infeasible_status::MOI.TerminationStatusCode = MOI.INFEASIBLE,
    exclude::Vector{Any} = Any[],
) where {T}
```

Return an object that is used to configure various tests.

Configuration arguments

- `atol::Real = Base.rtoldefault(T)`: Control the absolute tolerance used when comparing solutions.
- `rtol::Real = Base.rtoldefault(T)`: Control the relative tolerance used when comparing solutions.
- `optimal_status = MOI.OPTIMAL`: Set to `MOI.LOCALLY_SOLVED` if the solver cannot prove global optimality.
- `infeasible_status = MOI.INFEASIBLE`: Set to `MOI.LOCALLY_INFEASIBLE` if the solver cannot prove global infeasibility.
- `exclude = Vector{Any}`: Pass attributes or functions to `exclude` to skip parts of tests that require certain functionality. Common arguments include:
  - `MOI.delete` to skip deletion-related tests
  - `MOI.optimize!` to skip optimize-related tests
  - `MOI.ConstraintDual` to skip dual-related tests
  - `MOI.VariableName` to skip setting variable names
  - `MOI.ConstraintName` to skip setting constraint names

Examples

For a nonlinear solver that finds local optima and does not support finding dual variables or constraint names:

```
Config(
    Float64;
    optimal_status = MOI.LOCALLY_SOLVED,
```

```
    exclude = Any[
        MOI.ConstraintDual,
        MOI.VariableName,
        MOI.ConstraintName,
        MOI.delete,
    ],
)
```

source

MathOptInterface.Test.runtests – Function.

```
runtests(
    model::MOI.ModelLike,
    config::Config;
    include::Vector{String} = String[],
    exclude::Vector{String} = String[],
    warn_unsupported::Bool = false,
    exclude_tests_after::VersionNumber = v"999.0.0",
)
```

Run all tests in `MathOptInterface.Test` on `model`.

Configuration arguments

- `config` is a `Test.Config` object that can be used to modify the behavior of tests.
- If `include` is not empty, only run tests that contain an element from `include` in their name.
- If `exclude` is not empty, skip tests that contain an element from `exclude` in their name.
- `exclude` takes priority over `include`.
- If `warn_unsupported` is `false`, `runtests` will silently skip tests that fail with `UnsupportedConstraint` or `UnsupportedAttribute`. When `warn_unsupported` is `true`, a warning will be printed. For most cases the default behavior (`false`) is what you want, since these tests likely test functionality that is not supported by `model`. However, it can be useful to run `warn_unsupported = true` to check you are not skipping tests due to a missing `supports_constraint` method or equivalent.
- `exclude_tests_after` is a version number that excludes any tests to MOI added after that version number. This is useful for solvers who can declare a fixed set of tests, and not cause their tests to break if a new patch of MOI is released with a new test.

See also: `setup_test`.

Example

```
config = MathOptInterface.Test.Config()
MathOptInterface.Test.runtests(
    model,
    config;
    include = ["test_linear_"],
    exclude = ["VariablePrimalStart"],
    warn_unsupported = true,
    exclude_tests_after = v"0.10.5",
)
```

source

MathOptInterface.Test.setup_test – Function.

```
setup_test(::typeof(f), model::MOI.ModelLike, config::Config)
```

Overload this method to modify `model` before running the test function `f` on `model` with `config`. You can also modify the fields in `config` (e.g., to loosen the default tolerances).

This function should either return `nothing`, or return a function which, when called with zero arguments, undoes the setup to return the model to its previous state. You do not need to undo any modifications to `config`.

This function is most useful when writing new tests of the tests for MOI, but it can also be used to set test-specific tolerances, etc.

See also: runtests

Example

```
function MOI.Test.setup_test(
    ::typeof(MOI.Test.test_linear_VariablePrimalStart_partial),
    mock::MOIU.MockOptimizer,
    ::MOI.Test.Config,
)
    MOIU.set_mock_optimize!(
        mock,
        (mock::MOIU.MockOptimizer) -> MOIU.mock_optimize!(mock, [1.0, 0.0]),
    )
    mock.eval_variable_constraint_dual = false

    function reset_function()
        mock.eval_variable_constraint_dual = true
        return
    end
    return reset_function
end
```

source

MathOptInterface.Test.version_added – Function.

```
version_added(::typeof(function_name))
```

Returns the version of MOI in which the test `function_name` was added.

This method should be implemented for all new tests.

See the `exclude_tests_after` keyword of runtests for more details.

source

MathOptInterface.Test.@requires – Macro.

```
@requires(x)
```

Check that the condition x is `true`. Otherwise, throw an RequirementUnmet error to indicate that the model does not support something required by the test function.

Examples

```
@requires MOI.supports(model, MOI.Silent())
@test MOI.get(model, MOI.Silent())
```

source

MathOptInterface.Test.RequirementUnmet – Type.

```
RequirementUnmet(msg::String) <: Exception
```

An error for throwing in tests to indicate that the model does not support some requirement expected by the test function.

source