A Rust interface for SCIP

by Mohammed Ghannam (@mmghannam)

How it started?

- Late 2022: Learning Rust in my free time.
- Rust is great, it would be amazing to use SCIP from Rust.
- `good_lp` issue to add support for SCIP.

Why write a Rust interface for SCIP?

- No-overhead when binding to C.
- Memory safe and thread safe at compile time.
- No garbage collector.
- Great community and ecosystem.
- Great support for parallelism and concurrency.

First Step: bindings

`scip_sys`: (unsafe) Rust bindings to SCIP’s C API

- covers all of SCIP’s C API
- can be hard to work with.

Second Step - a Safe Wrapper

`russcip`: a safe and idiomatic Rust wrapper around `scip_sys`

Philosophy

- Use Rust's type system to enforce safety and correctness.
- Hide complexity and boilerplate code.

Current Features

- Easy access to SCIP through the `bundled` feature.
- Automatic memory management.
- Separate stages for model wrappers, avoiding many user errors at compile time. e.g. `focus_node()`
- Aim to reduce boilerplate code and improve usability.
- Simpler API for writing models (also through `good_lp`) and implementing callbacks.
- Unsafe access to SCIP's C API when needed through the `ffi` module.

russcip Guide

Modeling

$$
\begin{aligned}
&\text{maximize} \quad 3x_1 + 2x_2 \\
&\text{subject to:} \\
&\quad 2x_1 + x_2 \leq 100 \quad (c_1) \\
&\quad x_1 + 2x_2 \leq 80 \quad (c_2) \\
&\quad x_1, x_2 \geq 0 \text{ and integer}
\end{aligned}
$$

```rust
// Create model
let mut model = Model::default().maximize();

// Add variables
let x1 = model.add(var().int(0..).obj(3.).name("x1"));
let x2 = model.add(var().int(0..).obj(2.).name("x2"));

// Add constraints
model.add(cons().name("c1").coef(&x1, 2.).coef(&x2, 1.).le(100.));
model.add(cons().name("c2").coef(&x1, 1.).coef(&x2, 2.).le(80.));
```

Querying the solution

```rust
let solved_model = model.solve();

let status = solved_model.status();
println!("Solved with status {:?}", status);

let obj_val = solved_model.obj_val();
println!("Objective value: {}", obj_val);

let sol = solved_model.best_sol().unwrap();
let vars = solved_model.vars();

for var in vars {
    println!("{} = {}", var.name(), sol.val(&var));
}
```

```
    feasible solution found by trivial heuristic after 0.0 seconds, objective value 0.000000e+00
presolving:
(round 1, fast)       0 del vars, 0 del conss, 0 add conss, 3 chg bounds, 0 chg sides, 0 chg coeffs, 0
upgd conss, 0 impls, 0 clqs
(round 2, exhaustive) 0 del vars, 0 del conss, 0 add conss, 3 chg bounds, 0 chg sides, 0 chg coeffs, 2
upgd conss, 0 impls, 0 clqs
   (0.0s) symmetry computation started: requiring (bin +, int +, cont +), (fixed: bin -, int -, cont
-)
   (0.0s) no symmetry present (symcode time: 0.00)
presolving (3 rounds: 3 fast, 2 medium, 2 exhaustive):
 0 deleted vars, 0 deleted constraints, 0 added constraints, 3 tightened bounds, 0 added holes, 0
changed sides, 0 changed coefficients
 0 implications, 0 cliques
presolved problem has 2 variables (0 bin, 2 int, 0 impl, 0 cont) and 2 constraints
      2 constraints of type <varbound>
transformed objective value is always integral (scale: 1)
Presolving Time: 0.01
transformed 1/1 original solutions to the transformed problem space

 time | node  | left  |LP iter|LP it/n|mem/heur|mdpt |vars |cons |rows |cuts |sepa|confs|strbr|
dualbound    | primalbound  |  gap   | compl.
p 0.0s|     1 |     0 |     0 |     - | vbounds|   0 |   2 |   2 |   2 |   0 |  0 |   0 |   0 |
2.300000e+02 | 1.500000e+02 |  53.33%| unknown
* 0.0s|     1 |     0 |     2 |     - |    LP  |   0 |   2 |   2 |   2 |   0 |  0 |   4 |   0 |
1.600000e+02 | 1.600000e+02 |   0.00%| unknown
  0.0s|     1 |     0 |     2 |     - |   596k |   0 |   2 |   2 |   2 |   0 |  0 |   4 |   0 |
1.600000e+02 | 1.600000e+02 |   0.00%| unknown

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.02
Solving Nodes      : 1
Primal Bound       : +1.60000000000000e+02 (3 solutions)
Dual Bound         : +1.60000000000000e+02
Gap                : 0.00 %
Solved with status Optimal
Objective value: 160
t_x1 = 40
t_x2 = 20
```

Setting Parameters

SCIP has thousands of parameters. A full list can be found here

```
model.set_param("limits/softtime", 100.0);
```

Emphasis Modes

SCIP has meta-parameters that can be set to influence the solving process.

```rust
let mut model = Model::default();
model.set_heuristics(ParamSetting::Aggressive);
model.set_presolving(ParamSetting::Off);
model.set_separating(ParamSetting::Aggressive);
```

Plugins

Event Handlers

SCIP broadcasts many events during the solving process, callbacks can be registered to listen to these events.

Example

event handler to print node data, here.

Primal Heuristics

Primal heuristics are used to find feasible solutions during the solving process.

Example

Primal heuristic that rounds the current LP solution, here.

Branching Rules

Branching rules are used to select the next variable to branch on during the solving process (also enables custom branching).

Example

Most infeasible branching rule, here.

Separators

Separators can add valid inequalities to the model to tighten the LP relaxation.

Example

Clique separator for set partitioning problem, here.

Column Generation: Pricers

Column generation is a technique used to solve large-scale linear programming problems by solving a restricted master problem and generating new variables (columns) to add to the model.

Example

Pricer for the Cutting Stock Problem, here.

Future Work

Future Work: Simple Event Handlers

- Less boilerplate code for simple event handlers, by passing a closure and an event type.

```
let mut model = Model::default();
// ... some variables and constraints
model.set_callback(EventMask::NODE_FOCUSED, |model, event| {
    let node_number = model.focus_node().number();
    let node_depth = model.focus_node().depth();
    println!("Solved node number: {}, at depth: {}", node_number,
node_depth);
});
```

Future Work: More Safe Wrappers

SCIP supports many other callbacks, such as:

- Reader
- Presolver
- Cut selector

Many more API functions are available in SCIP, a full list can be found here.

Future Work: Modeling

Enable more powerful modeling features for the many constraint types available in SCIP through a generic procedural macro.

```
model.add(c!( 2 * x + y <= 10)); // linear constraint
model.add(c!( x * y <= 10)); // nonlinear constraint
model.add(c!( e ^ y <= 10)); // exponential constraint
model.add(c!( log(y) <= 10)); // logarithmic constraint
model.add(c!( sqrt(x) <= 10)); // square root constraint

model.add(c!( y -> x <= 10)); // indicator constraint
model.add(c!( (x + y == 10) && (x >= 5)  )); // AND constraint
model.add(c!( (x + y == 10) || (x >= 5)  )); // OR constraint
```

Future Work: Parallel plugins

Enable support for adding parallel plugins. They run on a separate thread and can only communicate with SCIP through an event handler and a message queue to modify the model.

Thank you for your attention!