ViskaSAT

Tamego

目次

第 1 章
アルゴリズム

1.1 基本設計

SAT ソルバのアルゴリズム部分だけを取り出してライブラリとして使えるようにする。

- **`Solver`** 様々なアルゴリズムのソルバの共通のインターフェース。このトレイトを持つものはロジックだけに集中し、Godot については何も触らない。
- **`SolverRunner`** `Solver` と Godot の橋渡し的役割をする。`Solver` を別スレッドで走らせ、ソルバの進捗を得たりソルバを制御したりと双方向のやりとりを可能にする。

可読性のためにソースを分割して記述する。

```
//| file: rust/viska-sat/src/lib.rs
pub mod lit;
pub mod clause;
pub mod cnf;
pub mod assignment;
pub mod event_handler;
pub mod solver;
pub mod solver_communicator;
```

1.2　基本的な型

1.2.1　リテラル

リテラルとは原子論理式それ自体もしくはその否定のどちらかである。とくに命題論理において原子論理式は命題変数だから、リテラルとは命題変数それ自体かその否定のいずれかである。

`var` が命題変数を表し、`negated` が否定されているかどうかを表す。

```
//| file: rust/viska-sat/src/lit.rs
pub struct Lit {
    pub var_id: usize,
    pub negated: bool
}
```

1.2.2　節

リテラルを OR で繋いだ論理式を**節**という。`Lit` の配列として表現する。 CDCL ソルバのために必要ならメタ情報を付けることを可能にした。

```
//| file: rust/viska-sat/src/clause.rs
use crate::lit::Lit;
pub struct Clause<Meta=()> {
    pub lits: Vec<Lit>,
    pub meta: Meta,
}
```

1.2.3　CNF（連言標準形）

節を AND で繋いだ構造をしている論理式を **CNF** という。`Clause` の配列として表現する。`num_vars` は変数の個数（最大の ID + 1）を表す。

```
//| file: rust/viska-sat/src/cnf.rs
use crate::clause::Clause;
pub struct Cnf {
    pub clauses: Vec<Clause>,
    pub num_vars: usize
}
```

1.2.4 割り当て

各変数に真偽値を対応付ける構造を**割り当て**という。真・偽・未割り当ての3つの値を取り得るので、`Option<bool>` 型を取ることで対応する。それの配列として割り当てを表現する。

```
//| file: rust/viska-sat/src/assignment.rs
pub struct Assignment {
    pub values: Vec<Option<bool>>
}
```

1.3 `Solver` トレイト

様々なアルゴリズムに共通するインターフェースを与えるために `Solver` トレイトを定める。 `solve()` は問題を解き、充足可能（`SatResult::Sat`）か充足不能（`SatResult::Unsat`）かをを返す。

`SolverReulst` は列挙型で定義する。

```
//| id: sol_solver-result
pub enum SatResult {
    Sat(Assignment),
    Unsat
}
```

`Solver` トレイトを定義する。 イベントの種類はソルバ依存なので、`Event` と関連型にして各ソルバが自由に決められるようにした。`EventHandler` については 1.5 節を参照のこと。

```
//| id: sol_solver-trait
pub trait Solver {
    type Event;
    type Error;
    type Handler: EventHandler<Event = Self::Event, Error =
Self::Error>;

    fn solve(&mut self) -> Result<SatResult, Self::Error>;
}
```

```
//| file: rust/viska-sat/src/solver.rs
use crate::{assignment::Assignment,
event_handler::EventHandler};
<<sol_solver-result>>
<<sol_solver-trait>>
```

1.4 `Solver` と `SolverRunner` 間の通信

`Solver` と `SolverRunner` 間の通信のために `SolverCommunicator` トレイトを定める。 これは以下の関数を持つ：

`SolverCommunicator::send_event()` `SolverRunner` にソルバのイベントを伝える。

`SolverCommunicator::recv_latest_control()` `SolverRunner` からの**最新の制御のみ**（`SolverControl`）を受けとる。スレッドをブロックする。

`SolverCommunicator::try_recv_latest_control()`
: `SolverRunner` からの**最新の制御のみ**を受けとる。スレッドをブロックしない。

`SolverControl` は全ソルバに共通するので、ここで列挙型で定義する。 停止と再開のみを想定しているが、今後増やすかもしれない。

```
//| id: sol_solver-control
pub enum SolverControl {
    Pause,
    Resume
}
```

また、`SolverCommunicator` に関連するエラーハンドリングのための列挙型を作る。

```
//| id: sol_solver-communicator-error
pub enum SolverCommunicatorError {
    SendFailed,
    ReceiveFailed,
}
```

イベントの種類はソルバ依存なので、`Solver` トレイトと同様に、`Event` と関連型にして各ソルバが自由に決められるようにした。

```
//| id: sol_solver-communicator
pub trait SolverCommunicator {
    type Event;

    fn send_event(&mut self, event: Self::Event) ->
Result<(), SolverCommunicatorError>;
    fn recv_latest_control(&mut self) ->
Result<SolverControl, SolverCommunicatorError>;
```

```
    fn try_recv_latest_control(&mut self) ->
Result<Option<SolverControl>, SolverCommunicatorError>;
}
```

```
//| file: rust/viska-sat/src/solver_communicator.rs
<<sol_solver-control>>
<<sol_solver-communicator-error>>
<<sol_solver-communicator>>
```

1.5　イベントハンドラー

`Solver` のイベントを処理する `EventHandler` トレイトを定義する。`handle_event()` でイベントを処理する。

```
//| id: evh_event-handler-trait
pub trait EventHandler {
    type Event;
    type Error;

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error>;
}
```

```
//| file: rust/viska-sat/src/event_handler.rs
<<evh_event-handler-trait>>
```

第 2 章

Godot

2.1 エントリーポイント

godot-rust API の主要部分を読み込む。

```
//| id: grl_godot-rust-api
use godot::prelude::*;
```

ファイル分割して記述したモジュールを読み込む。

```
//| id: grl_modules
mod tests;
```

空の構造体 `ViskaSATExtension` を作って、GDExtension 用のエントリーポイントにする。Godot とやりとりをする部分だから `unsafe` になっている。

```
//| id: grl_gdextension-entry-point
struct ViskaSATExtension;

#[gdextension]
unsafe impl ExtensionLibrary for ViskaSATExtension {}
```

```
//| file: rust/godot-rust/src/lib.rs
<<grl_godot-rust-api>>
<<grl_modules>>

<<grl_gdextension-entry-point>>
```

第 3 章
テスト

3.1　テスト用モジュール

　様々な内容の動作確認のために書いたコードを雑多にまとめておく。

```
//| file: rust/godot-rust/src/tests.rs
pub mod thread_channel_communication;
pub mod solver_trait;
```

3.2　スレッド間のチャネルの通信

　`Solver` と `SolverRunner` 間の通信の中核をなすチャネルについてテストしてみる。

　スレッドとチャネルのモジュールを読み込む。

```
//| id: tcc_modules
use std::thread;
use std::sync::mpsc;
use std::time::Duration;
```

3.2.1 `init`

フィールドの初期化をする。

```
//| id: tcc_init
fn init(base: Base<Control>) -> Self {
    Self {
        event_rx: None,
        ctrl_tx: None,
        is_pause: true,
        base
    }
}
```

3.2.2 `ready`

チャネルを立ててフィールドに代入したり、その他変数を用意する。

```
//| id: tcc_init_vars
let (event_tx, event_rx) = mpsc::channel::<u64>();
let (ctrl_tx, ctrl_rx) = mpsc::channel::<bool>();
self.event_rx = Some(event_rx);
self.ctrl_tx = Some(ctrl_tx);

let mut is_pause = self.is_pause;
```

そして、1 秒ごとに数字をカウントアップするスレッドを立てる。

```
//| id: tcc_thread
thread::spawn(move || {
    for val in 0..=100 {
        <<tcc_pause-handle>>
        event_tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
```

```
    }
});
```

ただし、一時停止のメッセージを受け取ったら再開のメッセージを受け取るまで停止する。

```
//| id: tcc_pause-handle
while let Ok(received) = ctrl_rx.try_recv() {
    is_pause = received;
}
while is_pause {
    let mut pause_flag = ctrl_rx.recv().unwrap();
    while let Ok(received) = ctrl_rx.try_recv() {
        pause_flag = received;
    }
    is_pause = pause_flag;
}
```

```
//| id: tcc_ready
fn ready(&mut self) {
    <<tcc_init_vars>>
    <<tcc_thread>>
}
```

3.2.3 `process`

そもそもチャネルが作られているか確認する。

```
//| id: tcc_check-channel
let (event_rx, ctrl_tx) = match (&self.event_rx,
&self.ctrl_tx) {
    (Some(rx), Some(tx)) => (rx, tx),
```

```
    _ => return,
};
```

もしデータがあるなら受け取る。

```
//| id: tcc_receive
if let Ok(received) = event_rx.try_recv() {
    godot_print!("{}", received);
}
```

決定ボタンが押されたらカウントアップの一時停止・再開をする。

```
//| id: tcc_pause-stop
let input = Input::singleton();
if input.is_action_just_pressed("ui_accept") {
    self.is_pause = !self.is_pause;
    godot_print!("is_pause: {}", self.is_pause);
    ctrl_tx.send(self.is_pause).unwrap();
}
```

```
//| id: tcc_process
fn process(&mut self, _delta: f64) {
    <<tcc_check-channel>>
    <<tcc_receive>>
    <<tcc_pause-stop>>
}
```

```
//| file: rust/godot-rust/src/tests/
thread_channel_communication.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<tcc_modules>>
```

```
#[derive(GodotClass)]
#[class(base=Control)]
struct ThreadChannelCommunication {
    event_rx: Option<mpsc::Receiver<u64>>,
    ctrl_tx: Option<mpsc::Sender<bool>>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for ThreadChannelCommunication {
    <<tcc_init>>
    <<tcc_ready>>
    <<tcc_process>>
}
```

3.3 Solver トレイト

1.3 節で定義した Solver トレイトをテストしてみる。3.2 節の一部を Solver トレイトを実装した構造体に置き換えて同様に動作することを確認する。

```
//| id: sot_modules
use viska_sat::{solver::{Solver, SatResult},
event_handler::EventHandler};
use std::sync::mpsc::{channel, Sender, Receiver,
TryRecvError};

use std::time::Duration;
use std::thread;
```

また、適当なエラーの型を作っておく。

```
//| id: sot_error-type
#[derive(Debug)]
struct TestError;
```

3.3.1 `TestHandler`

イベントハンドラーを作る。 3.2 節のそれとほぼ同じような内容にする。

```
//| id: sotth-decl
struct TestHandler {
    event_tx: Sender<u64>,
    ctrl_rx: Receiver<bool>,
    is_pause: bool
}
```

関連型を設定する。 `Event` は今の進捗、つまりカウントアップした数字を表せるように `u64` にした。 また、 `Error` は適当に作った `TestError` にしておく。

```
//| id: sotth_associated-types
type Event = u64;
type Error = TestError;
```

最新の制御メッセージを取る。

```
//| id: sotth_try-recv-latest
loop {
    match self.ctrl_rx.try_recv() {
        Ok(received) => self.is_pause = received,
        Err(TryRecvError::Empty) => break,
        Err(TryRecvError::Disconnected) => return
Err(TestError),
```

```
    }
}
```

停止中なら再開の制御メッセージが届くまで待機する。

```
//| id: sotth_recv-latest
while self.is_pause {
    self.is_pause = match self.ctrl_rx.recv() {
        Ok(val) => val,
        Err(_) => return Err(TestError),
    };

    <<sotth_try-recv-latest>>
}
```

イベントを送信する。

```
//| id: sotth_send-event
if self.event_tx.send(event).is_err() {
    return Err(TestError);
}
```

```
//| id: sot_test-handler
<<sotth-decl>>

impl EventHandler for TestHandler {
    <<sotth_associated-types>>

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error> {
        <<sotth_try-recv-latest>>
        <<sotth_recv-latest>>
        <<sotth_send-event>>
```

```
        Ok(())
    }
}
```

3.3.2 `DummySolver`

実際にソルバとしては機能しない、ただ数字をカウントアップするだけのダミー `DummySolver` を作る。

```
//| id: sotds_decl
struct DummySolver {
    handler: <DummySolver as Solver>::Handler,
}
```

関連型を設定する。これは `TestHandler` と同じ。

```
//| id: sotds_associated-types
type Event = u64;
type Error = TestError;
type Handler = TestHandler;
```

ソルバの中身を定義する。

```
//| id: sotds_solve
fn solve(&mut self) -> Result<viska_sat::solver::SatResult,
Self::Error> {
    for val in 0..=100 {
        if self.handler.handle_event(val).is_err() {
            return Err(TestError);
        }
        thread::sleep(Duration::from_secs(1));
    }
```

```
    Ok(SatResult::Unsat)
}
```

```
//| id: sot_dummy-solver
<<sotds_decl>>
impl Solver for DummySolver {
    <<sotds_associated-types>>
    <<sotds_solve>>
}
```

あとはこれを動かすだけ。 重要な部分だけピックアップする。

3.3.3 `ready`

チャネルを立てて、`EventHandler` に渡す。

```
//| id: sotr_start-channel
let (event_tx, event_rx) = channel::<u64>();
let (ctrl_tx, ctrl_rx) = channel::<bool>();
self.event_rx = Some(event_rx);
self.ctrl_tx = Some(ctrl_tx);
let handler = TestHandler {
    event_tx,
    ctrl_rx,
    is_pause: self.is_pause,
};
```

`DummySolver` を作って解き始める。

```
//| id: sotr_start-solving
let mut solver = DummySolver {
    handler
};
thread::spawn(move || {
```

```
    solver.solve().unwrap();
});
```

```
//| id: sot_ready
fn ready(&mut self) {
    <<sotr_start-channel>>
    <<sotr_start-solving>>
}
```

3.3.4 `process`

`process` は 3.2 節の実装をそのまま使用する。

```
//| file: rust/godot-rust/src/tests/solver_trait.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<sot_modules>>
<<sot_error-type>>
<<sot_test-handler>>
<<sot_dummy-solver>>


#[derive(GodotClass)]
#[class(base=Control)]
struct SolverTrait {
    event_rx: Option<Receiver<u64>>,
    ctrl_tx: Option<Sender<bool>>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for SolverTrait {
```

```
    fn init(base: Base<Control>) -> Self {
        Self {
            event_rx: None,
            ctrl_tx: None,
            is_pause: true,
            base
        }
    }

    <<sot_ready>>
    <<tcc_process>>
}
```