組込みシステム向け関数リアクティブプログラミング言語の内部DSLとしての実装

辻 裕太，森口 草介，渡部 卓雄

東京工業大学

June 26, 2020

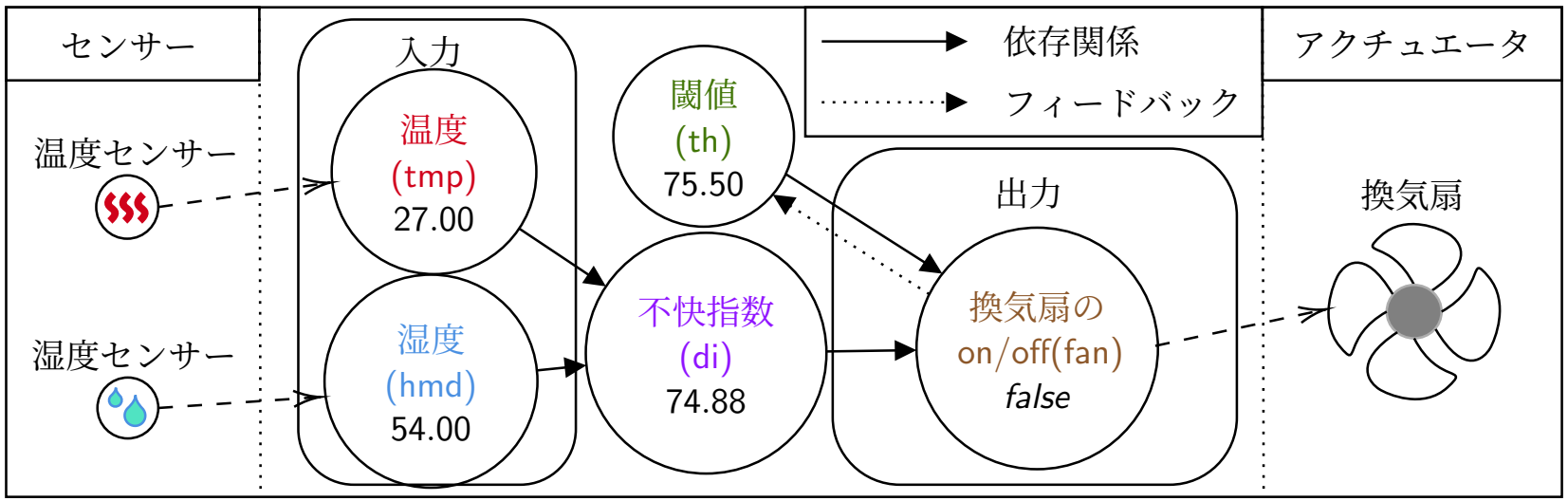
研究概要

- 目的
  - 限られた計算資源での動作，開発時の高い利便性，高速な実行を実現する FRP 言語の提供
- 手法
  - プログラミング言語 Rust 上の内部 DSL として動作する FRP 言語 LRFRP の設計及び処理系の実装
  - 手続きマクロを用いて LRFRP プログラムを解析し，他の Rust プログラムと協調して動作する Rust モジュールに変換
- 貢献
  - Rust が持つ言語機能や開発支援ツールを活用した，FRP による組込み開発の実現

関数リアクティブプログラミング（FRP）

- 時変値間の依存関係を宣言的に記述するプログラミングパラダイム
  - 時変値とは，時間に依存して変化する値を抽象化したもの
  - FRP では、時変値間の依存関係を宣言的に記述する
  - 入力時変値の変化に応じて、出力時変値が自動で計算される

- 例：センサーの値を用いたファンの制御

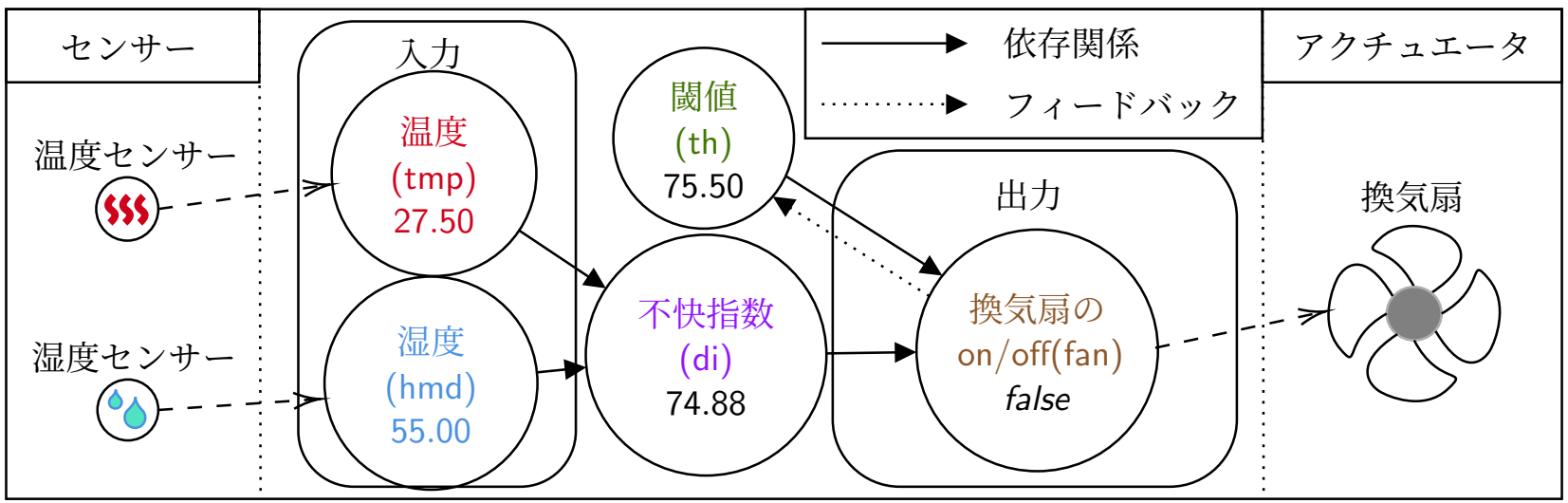



```
di ← 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
fan ← di ≥ th
th ← delay 75.0 + 0.5 ≺ 75.0 + if fan then -0.5 else 0.5
```

関数リアクティブプログラミング（FRP）

- 時変値間の依存関係を宣言的に記述するプログラミングパラダイム
  - 時変値とは，時間に依存して変化する値を抽象化したもの
  - FRP では、時変値間の依存関係を宣言的に記述する
  - 入力時変値の変化に応じて、出力時変値が自動で計算される

- 例：センサーの値を用いたファンの制御

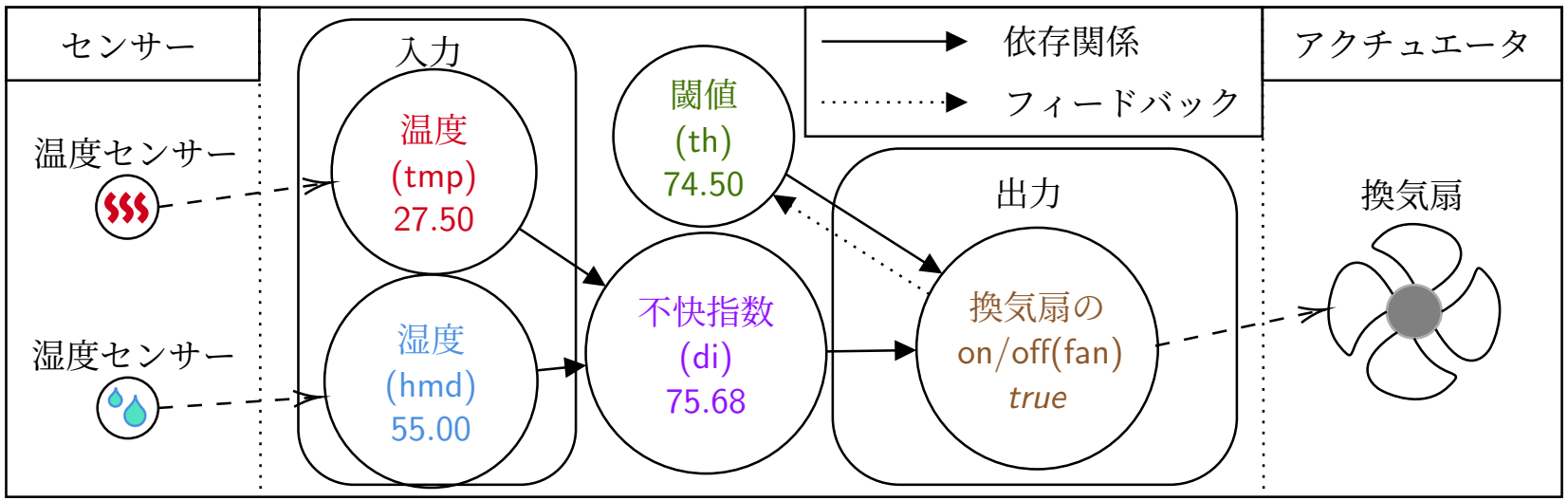



```
di ← 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
fan ← di ≥ th
th ← delay 75.0 + 0.5 ≺ 75.0 + if fan then -0.5 else 0.5
```

関数リアクティブプログラミング（FRP）

- 時変値間の依存関係を宣言的に記述するプログラミングパラダイム
  - 時変値とは，時間に依存して変化する値を抽象化したもの
  - FRP では、時変値間の依存関係を宣言的に記述する
  - 入力時変値の変化に応じて、出力時変値が自動で計算される

- 例：センサーの値を用いたファンの制御



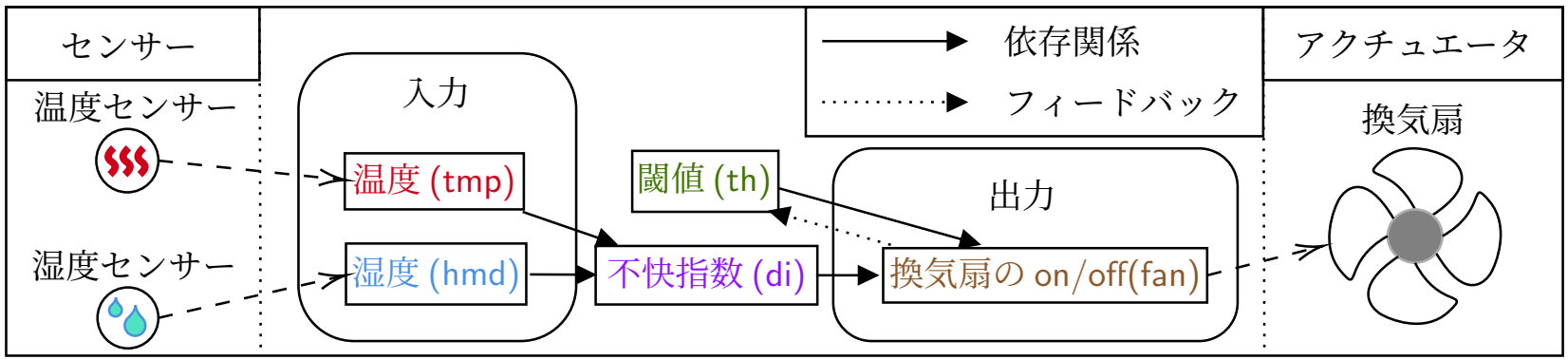

```
di ← 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
fan ← di ≥ th
th ← delay 75.0 + 0.5 ⇀ 75.0 + if fan then -0.5 else 0.5
```

Domain Specific Language（DSL）

- 特定の目的の達成のために作られたコンピュータ言語
  - SQL, Makefile, Verilog 等
- 大きく以下の 2 種類に分類される
  - 内部 DSL: 特定の言語に依存し，その上で動作するもの
  - 外部 DSL: 独自の処理系を持ち、言語自体が独立しているもの
- 本研究が提案する LRFRP は 内部 DSL
  - プログラミング言語 Rust をホスト言語とする
  - Rust の様々な恩恵を受けることができる

LRFRP

- Rust プログラム上に、独立した構文で記述する FRP 言語
- Rust モジュールに変換され，他の Rust プログラムから呼び出される

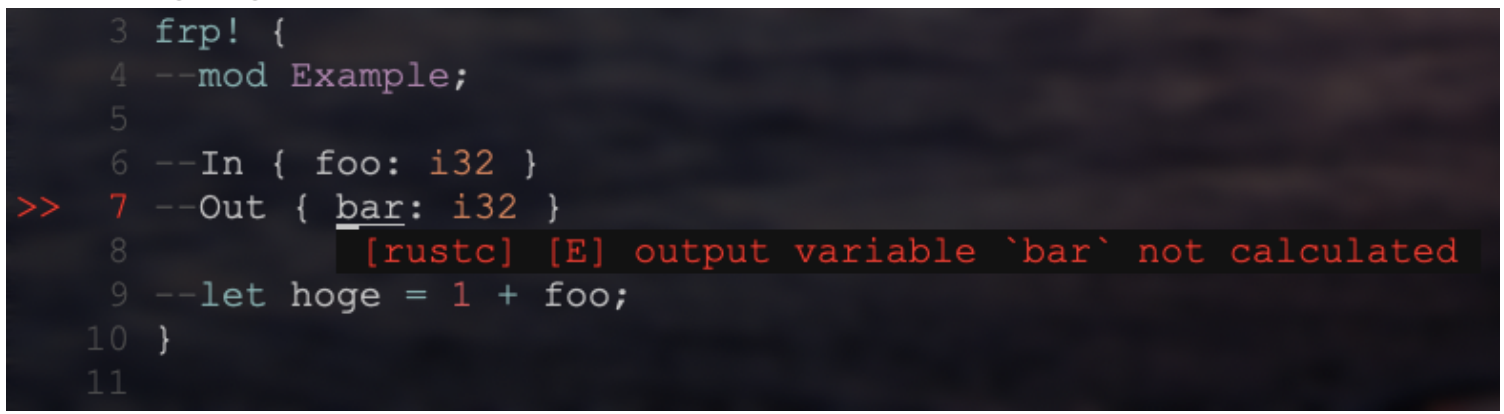



```
frp! {
    mod FanController;                                          // モジュール名

    Args { th_init: f32 }                                       // 初期化定数の定義

    In { tmp: f32, hmd: f32 }                                   // 入出力時変値の定義
    Out { fan: bool }                                           //

    fn calc_di(tmp: f32, hmd: f32) -> f32 =                     // 関数定義
        0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;   //

    let di = calc_di(tmp, hmd);                                 // 時変値間の依存関係
    let fan = di >= th;                                         //
    let th: f32 <- delay th_init -< th_init +                   //
        if fan then -0.5 else 0.5;                              //
}
```

ソースコード 1: LRFRP プログラムの例

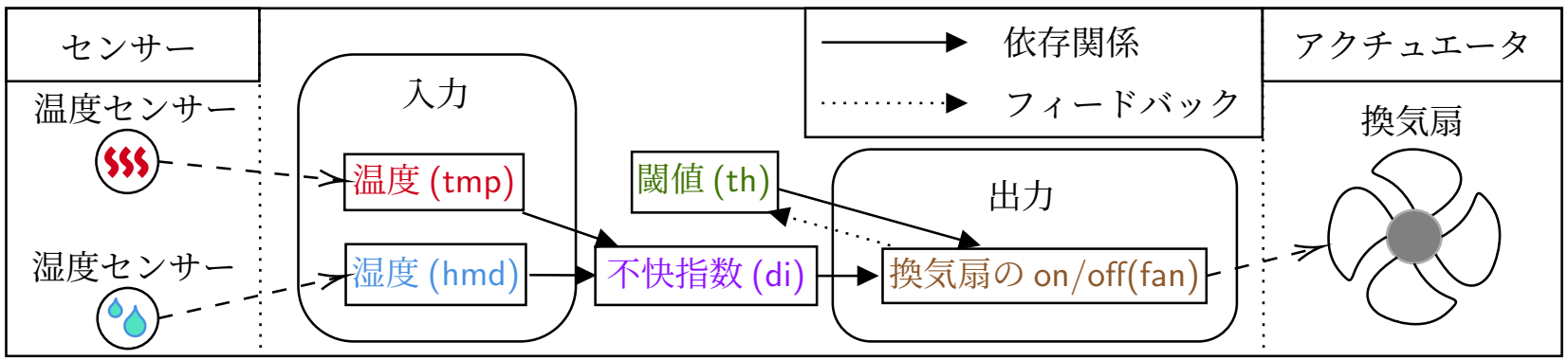
LRFRP の利点

- ホスト言語とは異なった独自の構文による，リアクティブシステムの簡潔な記述
- 最低限のパッケージ依存と軽量なコード生成によって実現される，組込み環境での動作
  - LRFRP 処理系は，ベアメタルでも利用できる，標準ライブラリのサブセット `libcore` のみに依存したコードを生成する
- ホスト言語が持つ性質の活用
  - 言語機能：型システム，リソース管理，ボローチェッカー
  - エコシステム：最適化，Cargo，開発環境
    - ★ 例：Language Server を用いた，開発中のエラーメッセージの表示



実行モデル

- サイクルでは，LRFRP プログラムに記述された全ての時変値が更新される
  - 他の Rust プログラムからサイクルが繰り返し実行される



LRFRP の時変値

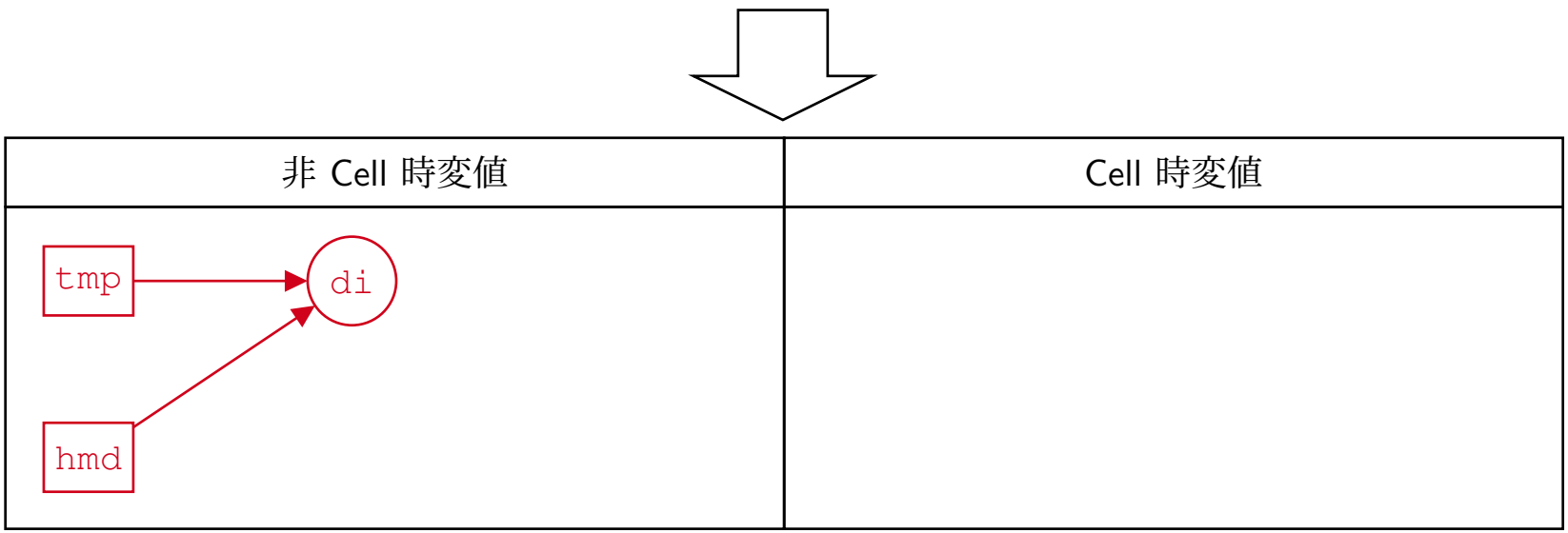
- LRFRP には二種類の時変値が存在する
  - 非 Cell 時変値：同サイクルにおける右辺式の評価結果で定義される
  - Cell 時変値：1 サイクル前における最右辺式の評価結果で定義される
    - ★ 最初のサイクルは `delay` キーワードの値を利用

```
1  frp! {
2      mod FanController;
3
4      Args { th_init: f32 }
5
6      // 入出力時変値
7      In { tmp: f32, hmd: f32 }
8      Out { fan: bool }
9
10     fn calc_di(tmp: f32, hmd: f32) -> f32 =
11         0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;
12
13     // 非 Cell 時変値の宣言
14     let di = calc_di(tmp, hmd);
15     let fan = di >= th;
16
17     // Cell 時変値の宣言
18     let th: f32 <- delay th_init -< th_init + if fan then -0.5 else 0.5;
19 }
```

時変値の計算順序

- 非 Cell 時変値は同サイクル，Cell 時変値は前サイクルの評価結果がそれぞれ割り当てられるようにする
  1. 2 種類の時変値間の依存関係をそれぞれ抽出
  2. 非 Cell 時変値は依存関係の順，Cell 時変値は依存関係の逆順で計算順序を割り振る (時変値間の依存関係のグラフは DAG になると仮定)

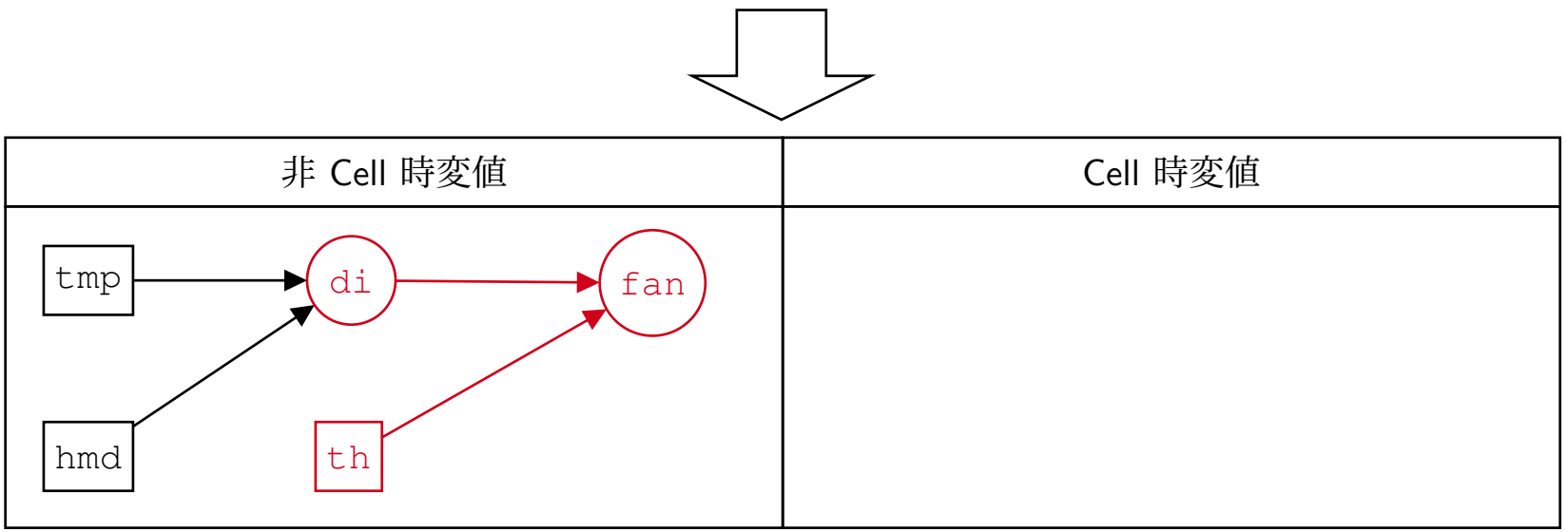
```
let di = calc_di(tmp, hmd);
let fan = di >= th;
let th:  f32 <- delay th_init -< th_init + if fan then -0.5 + 0.5;
```



時変値の計算順序

- 非 Cell 時変値は同サイクル，Cell 時変値は前サイクルの評価結果がそれぞれ割り当てられるようにする
  1. 2 種類の時変値間の依存関係をそれぞれ抽出
  2. 非 Cell 時変値は依存関係の順，Cell 時変値は依存関係の逆順で計算順序を割り振る (時変値間の依存関係のグラフは DAG になると仮定)

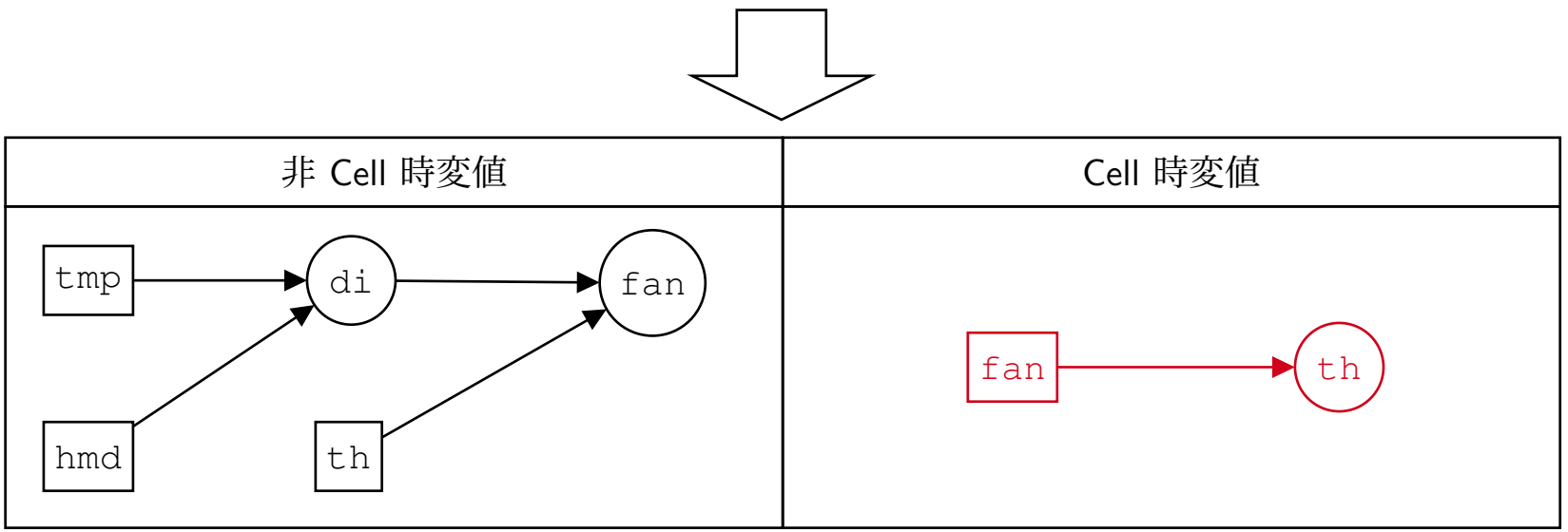
```
let di = calc_di(tmp, hmd);
let fan = di >= th;
let th:  f32 <- delay th_init -< th_init + if fan then -0.5 + 0.5;
```



時変値の計算順序

- 非 Cell 時変値は同サイクル，Cell 時変値は前サイクルの評価結果がそれぞれ割り当てられるようにする
  1. 2 種類の時変値間の依存関係をそれぞれ抽出
  2. 非 Cell 時変値は依存関係の順，Cell 時変値は依存関係の逆順で計算順序を割り振る (時変値間の依存関係のグラフは DAG になると仮定)

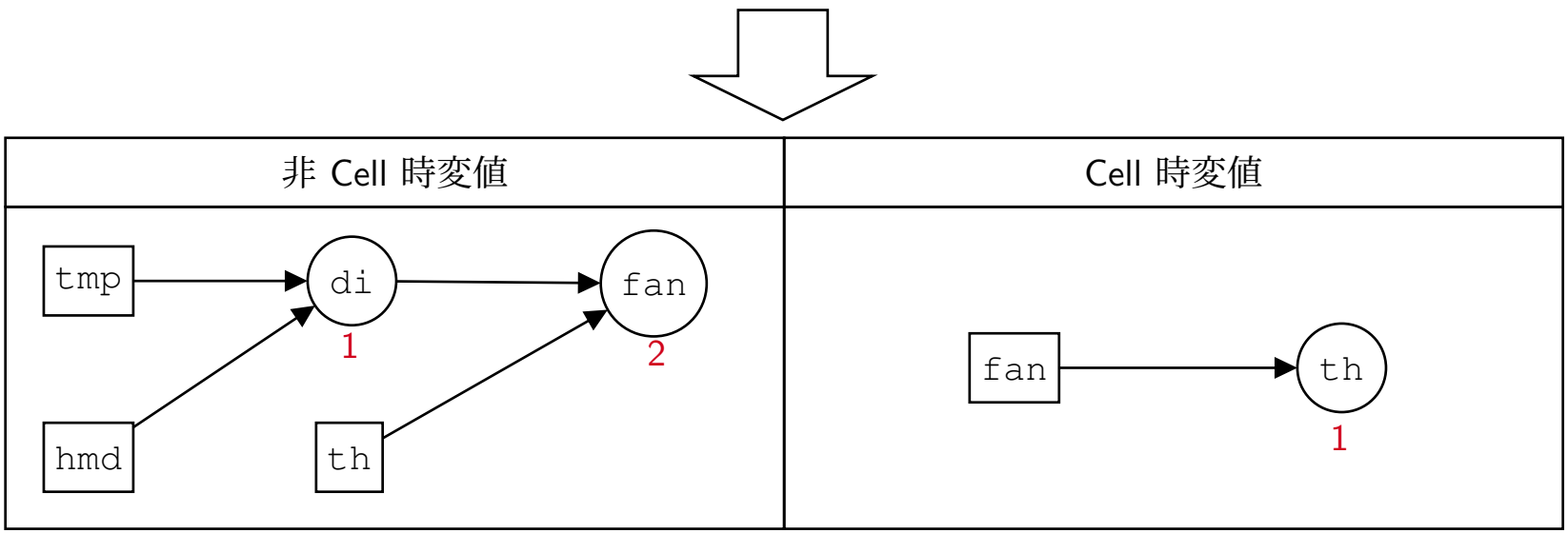
```
let di = calc_di(tmp, hmd);
let fan = di >= th;
let th:  f32 <- delay th_init -< th_init + if fan then -0.5 + 0.5;
```



時変値の計算順序

- 非 Cell 時変値は同サイクル，Cell 時変値は前サイクルの評価結果がそれぞれ割り当てられるようにする
  1. 2 種類の時変値間の依存関係をそれぞれ抽出
  2. 非 Cell 時変値は依存関係の順，Cell 時変値は依存関係の逆順で計算順序を割り振る (時変値間の依存関係のグラフは DAG になると仮定)

```
let di = calc_di(tmp, hmd);
let fan = di >= th;
let th:  f32 <- delay th_init -< th_init + if fan then -0.5 + 0.5;
```



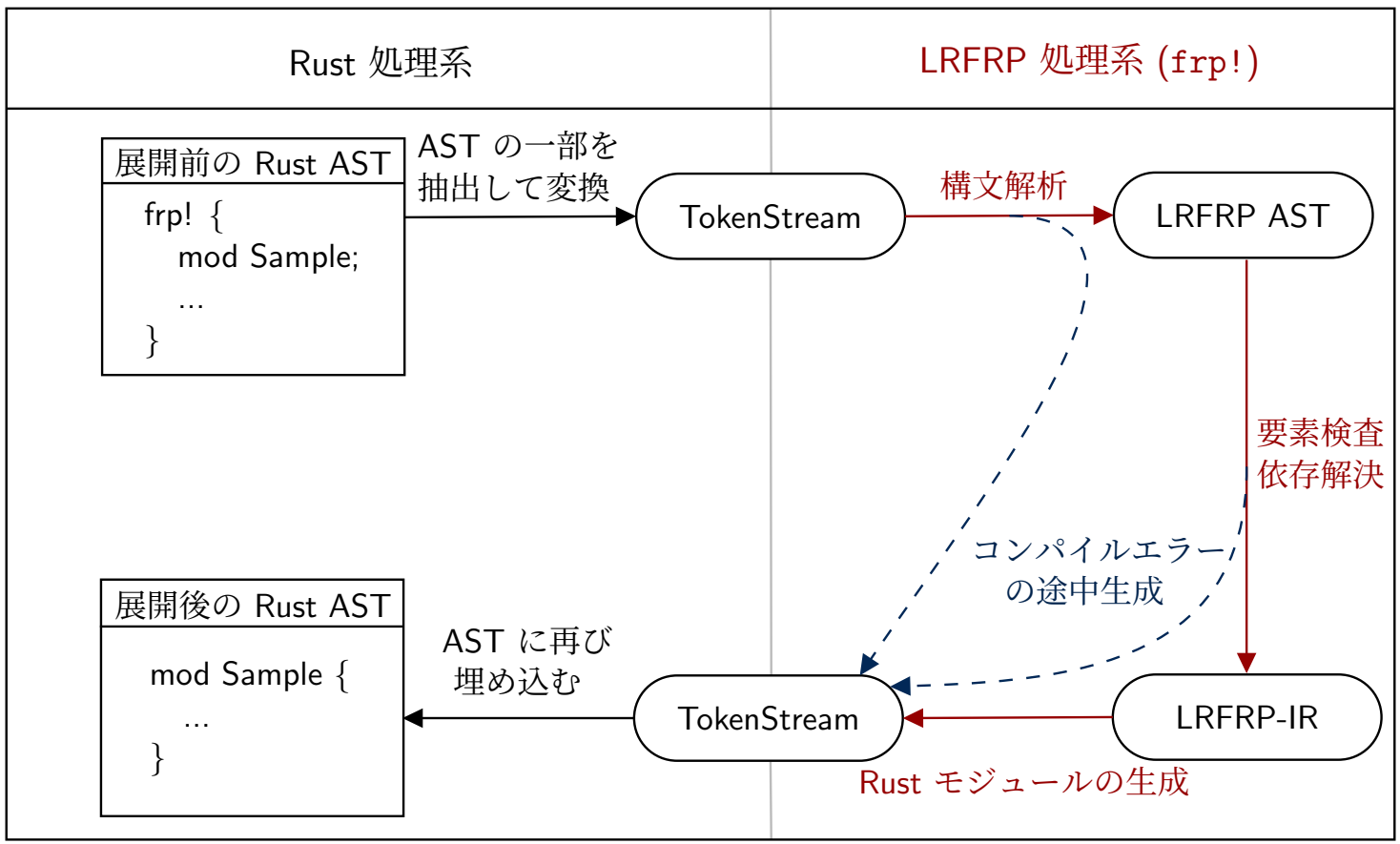
実装の概要

- LRFRP の処理系は，手続きマクロ `frp!` の実装として記述される
  - 手続きマクロ：Rust で利用できるマクロの一つ
  - Rust プログラムのコンパイル時における，任意の計算および
    トークンレベルでのプログラムの組み替えや生成が可能
    - ★ マクロの実装では，Rust プログラム上のトークン列を
      表現する構造体 (TokenStream) の組み替えを行う

- `frp!` は LRFRP プログラムを Rust モジュールへと変換
  - この実装は全て Rust で記述されており，計 2,500~3,000 行程度

LRFRP 処理系

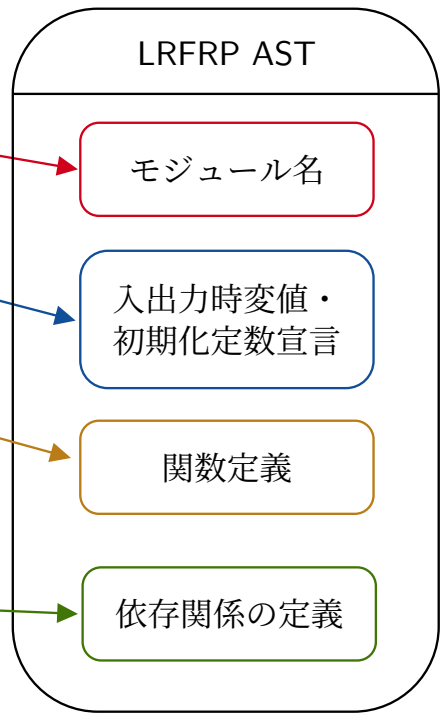
- LRFRP の処理系は，構文解析，意味解析，コード生成の 3 段階で TokenStream の組み替えを行う
- 手続きマクロの実装には任意のコンパイルエラーを発生させる機構がある
  - 前述のような詳細なエラーメッセージを生成可能



構文解析

- 再帰下降構文解析を行い，AST を作成する
  - Rust とは異なる独自の構文を持つことで，FRP に要求される副作用の制限が実現可能
  - 各トークンはプログラム上の位置情報を持つため，構文エラーの発生箇所を的確にプログラマに伝達できる

```
frp! {
    mod FanController;

    Args { th_init: f32 }
    In { tmp: f32, hmd: f32 }
    Out { fan: bool }

    fn calc_di(tmp: f32, hmd: f32) -> f32
        = 0.81 * tmp + 0.01 * hmd *
            (0.99 * tmp - 14.3) + 46.3;

    let di = calc_di(tmp, hmd);
    let fan = di >= th;
    let th: f32 <- delay th_init -< th_init +
        if fan then -0.5 else 0.5;
}
```



Rust プログラム上の手続きマクロ `frp!` に記述された LRFRP プログラム

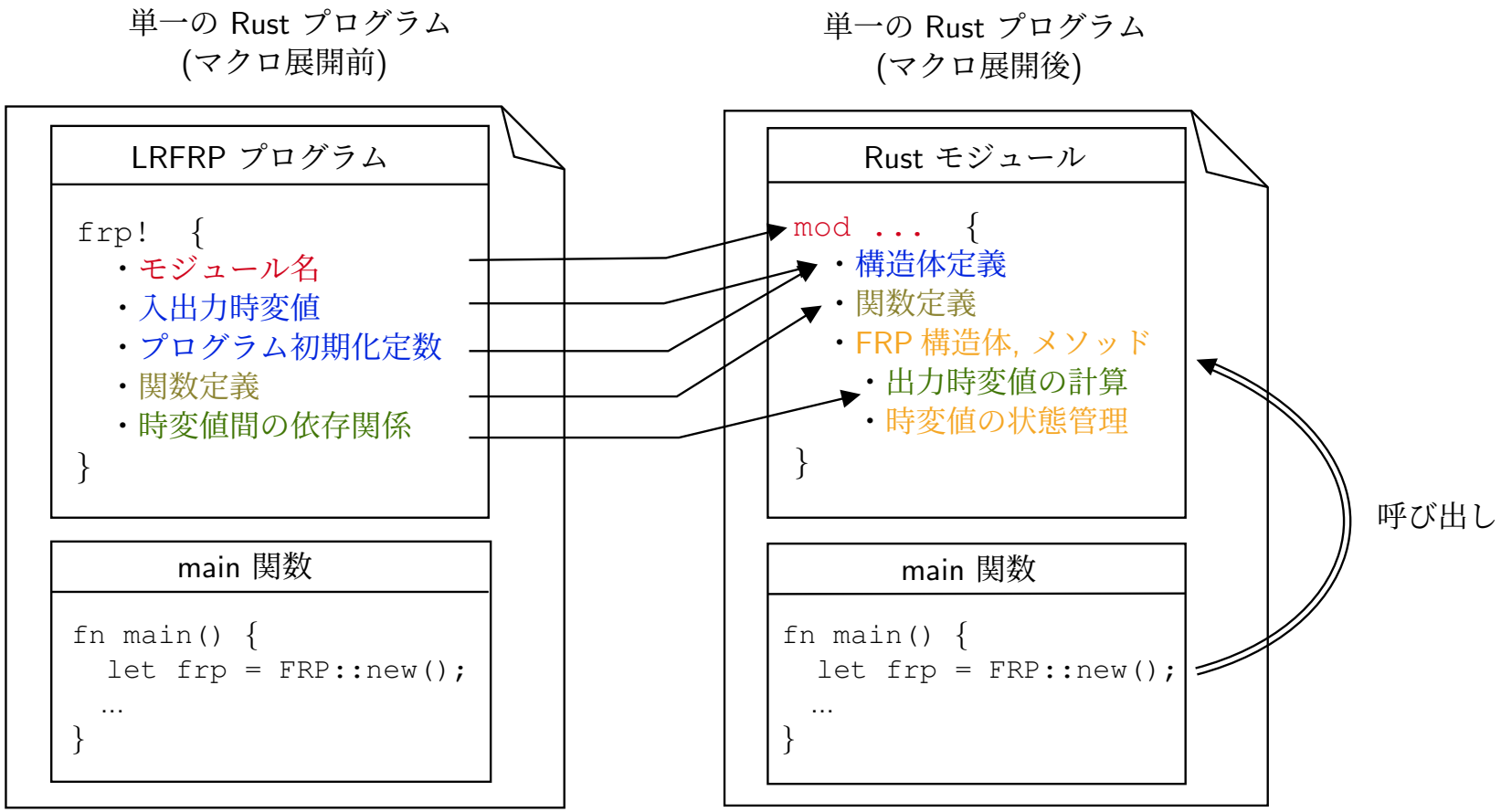
意味解析

- 以下の手順によって，LRFRP の AST から LRFRP-IR を生成
  1. 環境抽出
     - ★ 全ての時変値及び関数の宣言情報を収集
     - ★ LRFRP の構文要素が過不足なく宣言されているかを確認
  2. 依存抽出
     - ★ プログラム上の時変値間の依存関係を抽出
  3. 時変値更新の順序決定
     - ★ 得られた依存関係をもとに，時変値の計算順序を決定

- 構文要素の過不足，重複した時変値宣言，循環した依存関係の記述が存在する不正なプログラムに対しても，構文解析と同様に詳細なエラーメッセージを出力する

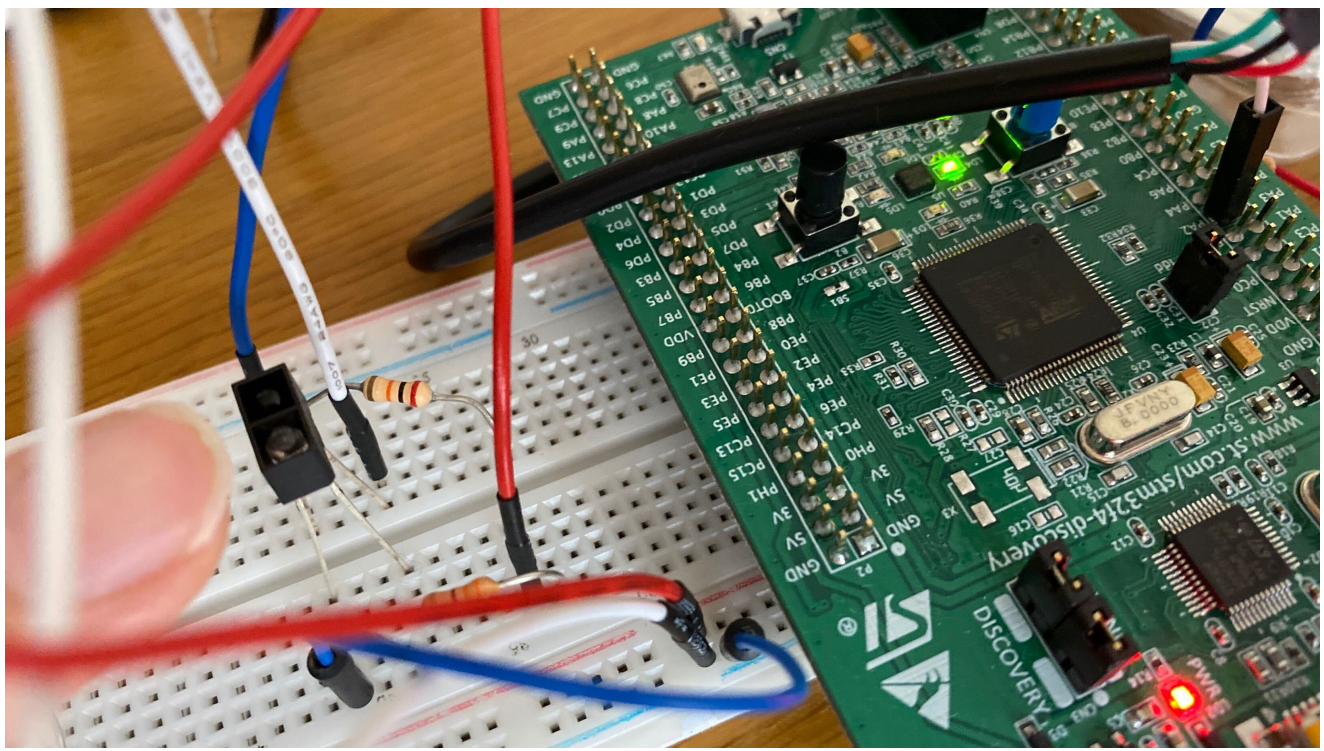
コード生成

- LRFRP-IR から Rust モジュールを生成
- このモジュールは Rust プログラムの一部として，他の Rust のプログラムから利用できる



ケーススタディ

- 冷蔵庫の「開けっ放し」をフォトトランジスタで検知する簡易的なシステム FridgeAlarm の実装
- 環境
  - マイコン：STM32F407VG
  - コンパイラ：rustc 1.45.0-nightly (fa51f810e 2020-04-29)



LRFRP による実装

- フォトトランジスタから A/D 変換によって電圧を取得し，これが閾値を超えてから一定時間経過した時に警告を行う

```
frp! {
    mod FridgeAlarm;

    Args {
        init_time: u32,
        sensor_th: u32,
        alert_time: u32,
    }

    In {
        cur_time: u32,
        sensor: u32,
    }

    Out {
        alert: bool
    }

    // 経過時間の計測
    let prev_time: u32 <- delay init_time -< cur_time;
    let dt = cur_time - prev_time;
    let prev_elapsed: u32 <- delay 0 -< elapsed;
    let elapsed = if closed then 0 else dt + prev_elapsed;

    // 冷蔵庫の扉の状態
    let closed = sensor <= sensor_th;

    // アラームの状態
    let alert = elapsed >= alert_time;
}
```

比較実験

- LRFRP を用いた Rust プログラムと，LRFRP が行う計算を直接手続き的に記述したプログラムについて，生成されるバイナリのサイズの比較を行った (size コマンドによる)

- LRFRP によるシステムの記述によってコードの行数は増えてしまうものの，生成されるバイナリサイズはほぼ変わらないことが分かった

	行数	text(B)	data(B)	bss(B)
LRFRP 有り	131	2,784	32	8
LRFRP 無し	87	2,772	32	8

関連研究

- Emfrp[1)]
  - 外部 DSL としての組込み向け FRP 言語
  - LRFRP と同様に，時変値の依存関係から計算順序を求めて C 言語のプログラムを生成する
  - Emfrp とは異なり，LRFRP はホスト言語の持つ機能を直接利用できる
    - ★ Language Server を活用した，開発中におけるエラーメッセージの表示
    - ★ Rust のクレートとして利用できることによる，導入コストの削減

- Carboxyl[2)]
  - Rust 上の内部 DSL としての FRP 言語
  - 時変値間の依存関係を実行時に変えることができる
    - ★ LRFRP は静的に計算順序を決定するため，依存関係は不変である必要がある
  - 組込み開発を前提として作られていないため，計算資源の限られた環境で動かすのは困難がある

関連研究

- Hae[3] [Wang 2019]
  - Haskell 上の，Deep Embedded な内部 DSL としての組み込み向け FRP 言語
  - Hae プログラムは C++ のコードとして生成される

  ⇒ FRP ランタイムの記述でホスト言語の機能を使えない
    - LRFRP は Rust のモジュールとして展開されるため，ホスト言語の持つメモリ安全性の保証や型システムを利用してランタイムを記述できる

まとめ

- 結論
  - 内部 DSL としての FRP 言語 LRFRP を設計及び実装した
  - Rust が持つ言語機能や開発支援ツールを活用した，FRP による組込み開発が実現できたことを確認した
- 今後の課題
  - メモリ使用量やバイナリサイズの静的な見積もり
  - 型システムの導入
  - LRFRP プログラムの再利用性の確保

参考文献 I


Kensuke Sawada and Takuo Watanabe.
Emfrp: A functional reactive programming language for small-scale embedded systems.
In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pp. 36–44, New York, NY, USA, 2016. Association for Computing Machinery.


edibopp/carboxyl.
`https://github.com/edibopp/carboxyl`.


Wang Sheng and Takuo Watanabe.
Functional reactive EDSL with asynchronous execution for resource-constrained embedded systems.
In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Vol. 850 of *Studies in Computational Intelligence*, pp. 171–190. Springer, Aug. 2019.

追加資料

FridgeAlarm の実行プログラム

```rust
#[cortex_m_rt::entry]
fn main() -> ! {
    // LED や ADC の初期化
    ...

    // サイクルの実行
    use FridgeAlarm::*;

    // FRP 構造体のインスタンスを作成
    let mut frp = FRP::new(Args {
        init_time,
        sensor_th: 1500,
        alert_time: 10_000,
    });

    loop {
        // センサー値の取得
        let sample = adc.convert(&pa1, SampleTime::Cycles_480);
        let bits = adc.sample_to_millivolts(sample);

        // 現在の時間の取得
        let time = free(|cs| ELAPSED_MS.borrow(cs).get());

        // 入力時変値の設定
        let input = In {
            cur_time: time,
            sensor: bits as u32,
        };

        // 実行, 出力時変値の取得
        frp.run(&input);
        let output = frp.sample().unwrap();

        // LED の点灯
        set_bs12(output.alert);
    }
}
```

FridgeAlarm の変換後の Rust モジュール (1/2)

```rust
#[allow(non_snake_case)]
mod FridgeAlarm {
    #[derive(Clone, Default)]
    pub struct In {
        pub cur_time: u32,
        pub sensor: u32,
    }
    #[derive(Clone, Default)]
    pub struct Out {
        pub alert: bool,
    }
    #[derive(Clone, Default)]
    pub struct Args {
        pub init_time: u32,
        pub sensor_th: u32,
        pub alert_time: u32,
    }
    #[derive(Clone, Default)]
    struct Cell {
        prev_time: u32,
        prev_elapsed: u32,
    }
    #[derive(Clone, Default)]
    pub struct FRP {
        running: bool,
        output: Out,
        args: Args,
        cell: Cell,
    }
```

FridgeAlarm の変換後の Rust モジュール (1/2)

```
    impl FRP {
        #[inline]
        pub fn new(args: Args) -> Self {
            FRP {
                running: false,
                output: Out::default(),
                args,
                cell: Cell::default(),
            }
            .cell_initializations()
        }
        #[inline]
        fn cell_initializations(mut self) -> Self {
            self.cell.prev_time = self.args.init_time;
            self.cell.prev_elapsed = 0;
            self
        }
        #[inline]
        pub fn sample(&self) -> core::option::Option<&Out> {
            if self.running {
                Some(&self.output)
            } else {
                None
            }
        }
        pub fn run(&mut self, input: &In) {
            self.running |= true;
            let dt = input.cur_time - self.cell.prev_time;
            let closed = input.sensor <= self.args.sensor_th;
            let elapsed = if closed {
                0
            } else {
                dt + self.cell.prev_elapsed
            };
            self.output.alert = elapsed >= self.args.alert_time;
            self.cell.prev_time = input.cur_time;
            self.cell.prev_elapsed = elapsed;
        }
    }
}
```