内部 DSL としての関数リアクティブプログラミング言語の実装手法

辻 裕太・森口 草介・渡部 卓雄 （東京工業大学）

概要

関数リアクティブプログラミング（FRP）は，時間に依存して値が変化する時変値における依存関係を，副作用のない計算を用いて宣言的に記述するプログラミングパラダイムである．本研究は，Deep Embeddedな内部DSLとしてのFRP言語LRFRPとその実装を提案する．ホスト言語として用いるRustは静的なメモリ安全性の検査を提供し，さらに組み込み開発への応用も進んでいる．そのためLRFRPはC/C++をベースとしたFRP言語に比べ，よりメモリ安全な組み込み開発を提供することも期待できる．

関数リアクティブプログラミング（FRP）

時間によって変化する**時変値**と呼ばれる値の間の依存関係を，純粋関数を用いて宣言的に記述するプログラミング パラダイム．
ウェブサイトやゲームのように，時間によって逐次的に変化する入力に対し，応答的に振る舞う処理が要求されるソフトウェアの開発に適している．

LRFRP

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．
手続きマクロを用いて LRFRP プログラムの変換を行い，他の Rust プログラムと協調して動作する Rust モジュールを生成する．
処理系の実装は 3,000 行程度．

既存研究との比較

Emfrp （Sawada et al. 2016）はリソースの制限された環境においても動作する組み込み向け FRP 言語であり， C言語をホスト言語とした外部 DSL である．

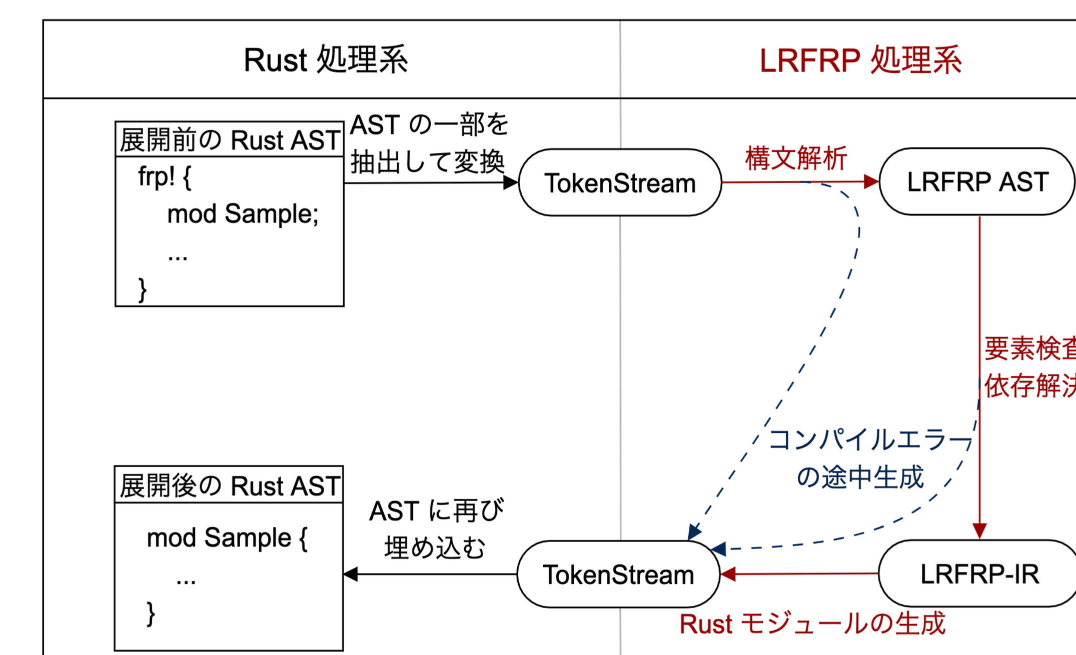
ここでは，後述する FanController を実装した Emfrp プログラム及び LRFRP プログラムを用いて実行速度の比較を行った．
両プログラムで時変値更新を 10^9 回行う試行を30 回ずつ行い， time コマンドを用いてユーザーCPU時間を計測し，それらの平均を求めて比較した．

	Emfrp	LRFRP
OS	macOS Mojave	
CPU	1.6GHz Intel Core i5	
ホスト言語のコンパイラ	clang (1001-0.46.4)	rustc 1.42.0-nightly (3a3f4a7cb 2019-12-28)
最適化オプション	-O2	-C opt-level=3 (Release モード)
実行速度 [sec/10^9cycle]	8.68	6.23

LRFRP の処理系

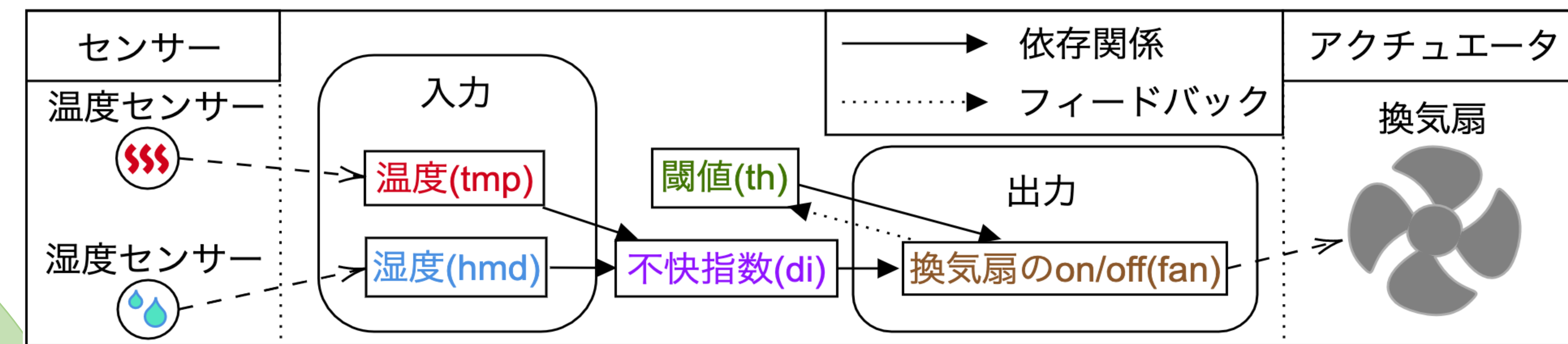
LRFRP プログラムは右図のように，Rust 処理系を経由してコンパイルされる．
Rust 上のトークン列を表現する TokenStream を解析し，Rust のモジュールを表現する TokenStream に再変換する．

1. 構文解析・・・LRFRP に対して再帰下降構文解析を行い，AST を生成する．
2. 意味解析
   2.1 要素検査・・・ モジュール名や時変値宣言等の構文要素が過不足なく存在しているか確認する．
   2.2 依存解決・・・ 時変値間の依存関係から，トポロジカルソートによってそれらの計算順序を決定する．
3. コード生成・・・Rust の構文として有効な， Rust モジュールを表現した TokenStream へと変換する．



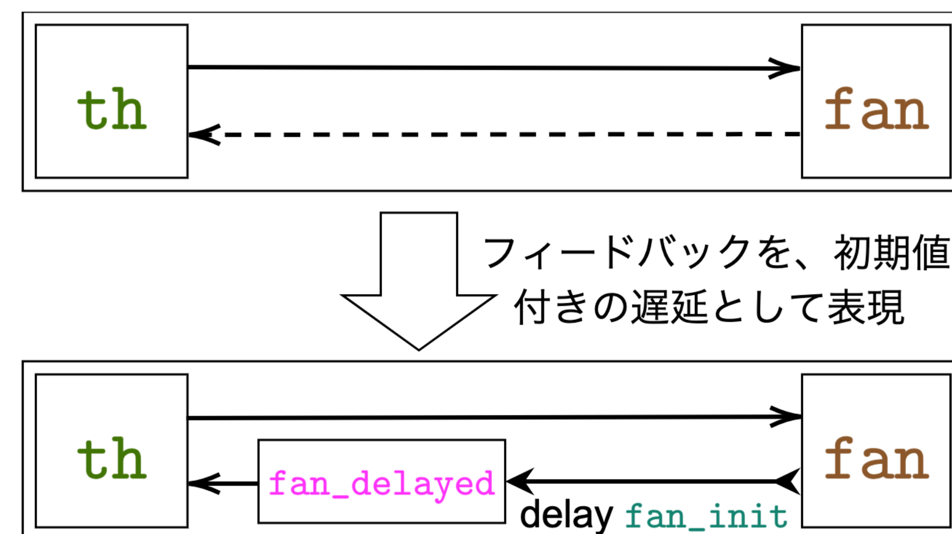
LRFRP によるプログラムの例

センサーの値に基づいて，ファンの on/off をヒステリシス制御によって計算する
プログラム FanController を考える．

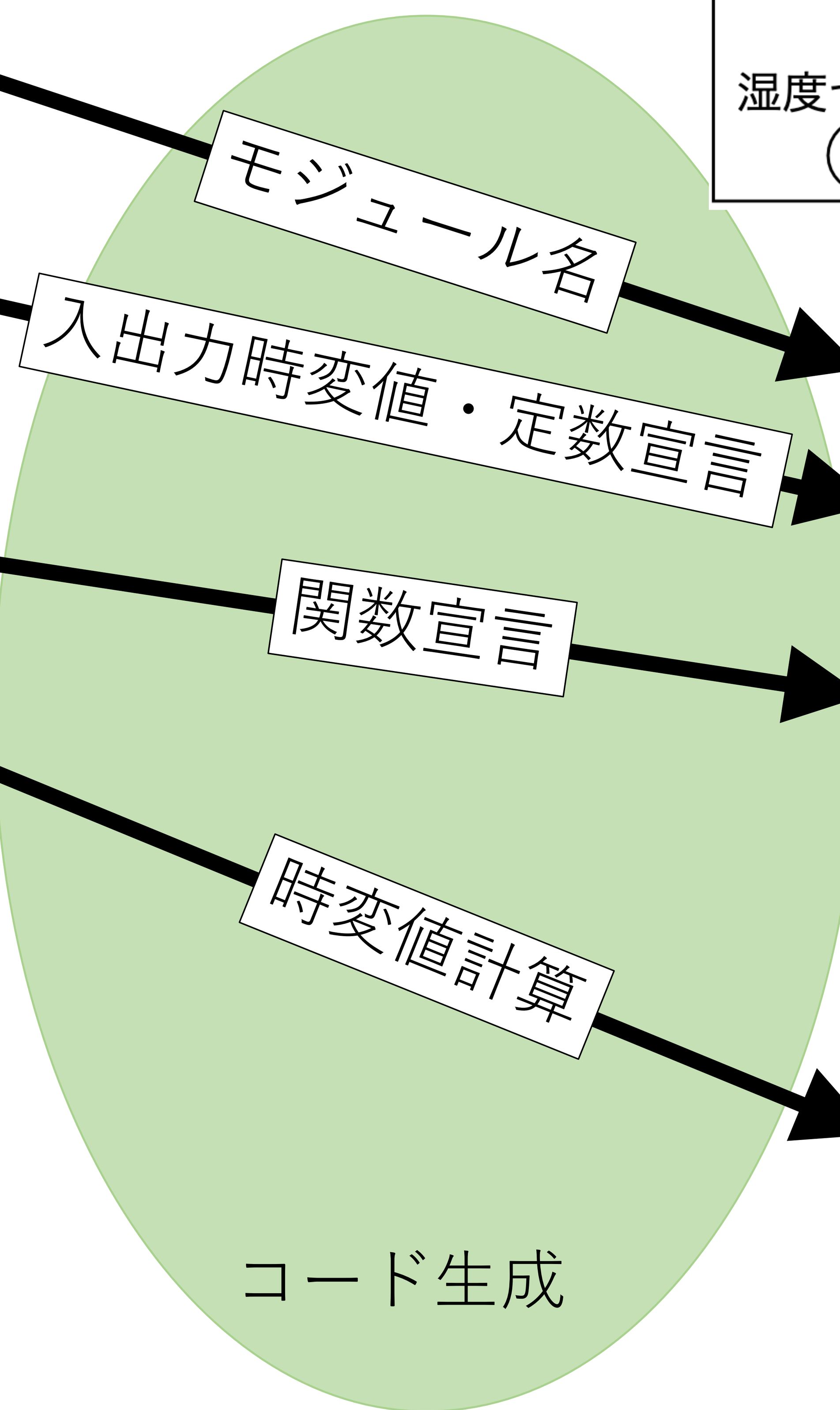


FanController の模式図

```
frp! {
  mod FanController;

  In { hmd: f32, tmp: f32 }
  Out { fan: bool }
  Args { fan_init: 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 fan_delayed: bool <- delay fan_init -< fan;
  let th = 75.0 + if fan_delayed then -0.5 else 0.5;
}

fn main() {
  use FanController::*;
  let args = FRP::Args { ... };
  let mut frp = FRP::new(args);
  loop {
    let input = FRP::In { ... };
    frp.run(&input);
    let output = frp.sample().unwrap();

    // output を用いて計算
    ...
  }
}
```



LRFRP を用いた FanController の実装




```
mod FanController {
  pub struct In { hmd: f32, tmp: f32 }
  pub struct Out { fan: bool }
  pub struct Args { fan_init: bool }

  fn calc_di(tmp: f32, hmd: f32) -> f32 { ... }

  struct Cell { fan_delayed: bool }
  pub struct FRP { ... }

  impl FRP {
    // コンストラクタ
    pub fn new(args: Args) -> Self { ... }
    // セル時変値（遅延によって新しく作られる時変値）の初期化
    fn cell_initialization(mut self) -> Self { ... }

    // 出力時変値の値の取得
    pub fn sample(&self)-> Option<&Out> { ... }
    // 入力時変値を用いた，出力時変値の計算
    pub fn run(&mut self, input: &In) { ... }
  }
}

fn main() { ... }
```

LRFRP プログラムから生成される Rust モジュール

生成コードの行数

LRFRP で記述されたいくつかのサンプルプログラムから
変換されるコードの行数を計測した．

サンプルプログラム	A	B	C
LRFRP プログラム frp! { ... } の行数	15	35	718
生成された Rust モジュールの行数 （フォーマッタ（Rustfmt）によって整形済）	62	71	757

今後の課題

・型システムの導入
　現在は Rust の型検査器を利用しているが，コード生成後の検査になるため，エラーの可読性が落ちる．独自の型システムと検査器を実装することで，この問題が解決できる．

・モジュール性の確保
　Rust の手続きマクロは他のマクロ展開に干渉できないため，別の LRFRP プログラムが持つ関数や構造体を直接利用できない．

・組み込み開発の実験
　LRFRP が生成するコードは組み込み向け標準ライブラリ core のみに依存しているため，組み込み開発への応用が期待できる．