Vol.2020-EMB-54 No.3
2020/6/26

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[1,a)]　森口 草介[1,b)]　渡部 卓雄[1,c)]

概要：関数リアクティブプログラミング (FRP) は，時間とともに変化する値（時変値）間の依存関係を副作用のない式で表すことで，リアクティブな動作の宣言的な記述を支援するプログラミングパラダイムである．本研究では，組込みシステム向けに設計された FRP 言語 LRFRP を提案する．LRFRP はプログラミング言語 Rust 上の内部 DSL であり，Rust のマクロ機能を用いて実装されている．Rust プログラム中に埋め込まれた LRFRP のプログラムは，コンパイル時に Rust のモジュールに変換される．このモジュールは極めて限られたライブラリにのみ依存し，また Rust の機能を制限しない．そのため，組込みシステムをターゲットとする場合など，利用できるライブラリが限定された状況でも動作可能である．本稿では LRFRP の概要と実装方式について述べ，例を通してその有効性について議論する．

キーワード：関数リアクティブプログラミング, DSL, Rust, 組込みシステム

1. はじめに

組込みシステムや GUI システムは，センサーの値やマウスの位置などの連続的に変化する入力，あるいはボタンの押下などの離散的な入力に対して反応（出力と自身の状態の更新）を行う．このようなシステムは**リアクティブシステム**と呼ばれる．

リアクティブシステムでは，入力が与えられる順序やタイミングは実行時にならないとわからない．このような前提でリアクティブな動作を実装するもっとも簡単な手法はポーリングであるが，その他にも特定のイベントに対応したイベントハンドラを呼び出す機構を用いるイベント駆動型プログラミングや，Promise や Future といった非同期計算にもとづく機構による手法などがある．多くの言語では，これらの手法を用いる際にはコールバックと呼ばれる関数（あるいはメソッド）を用いて実現する．コールバックによる実現では，ひとまとまりの処理がいくつかのコールバックに分断され，プログラムの見通しが悪くなるという欠点がある（callback hell などと呼ばれる）．**関数リアクティブプログラミング (Functional Reactive Programming, FRP)** は，時間とともに変化する値を抽象化した**時変値 (time-varying value)** と呼ばれる概念を導入し，それらの間の依存関係を副作用のない式（および関数）を用いて記述することで，リアクティブな動作の宣言的な記述を支援するプログラミングパラダイムである．これによって，callback hell を生じずに見通し良くプログラムを構成することが可能になる．

FRP は Fran (Functional Reactive Animation) [5] において提案され，アニメーションの他にロボット制御 [8] や GUI [4] などへの応用が行われている． 汎用の FRP ライブラリとしては Sodium [2]，Yampa [3] が挙げられる．これらの FRP ライブラリではメモリや CPU といった計算資源が比較的豊富な環境を想定しており，組込みシステムのように制限が多い環境には適用が難しい．一方で組込みシステム向けにいくつかの言語が提案されている [6, 7] ものの，環境を固定していたり汎用性に欠けているといった欠点がある．

本研究では, Rust をホスト言語とする FRP 言語 LRFRP を提案する．LRFRP はホスト言語との強い協調性，処理系による生成コードの最適化，独自の構文によって様々な利点を獲得している．

本論文は次のように構成される．次節で LRFRP について概要を述べ，続く第 3 節で内部 DSL としての実装について説明する．そしてケーススタディにもとづいた実装の評価を第 4 節で行う．第 5 節で関連研究との比較を行い，

[1] 東京工業大学情報理工学院情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology, Meguro, Tokyo, 152-8552, Japan
[a)] g2@psg.c.titech.ac.jp
[b)] chiguri@acm.org
[c)] takuo@acm.org

第 6 節で結論と今後の課題について述べる．

2. LRFRP

2.1 概要

LRFRP は Rust プログラム上に記述される内部 DSL としての FRP 言語である．LRFRP は独自の構文を持ち，時変値の宣言順序が自由であったり，依存する他の時変値に対する後方参照を許したり，といった特徴を持ち合わせている．これにより，フィードバックを含むようなリアクティブシステムにおいても簡潔にプログラムを記述することができる．

また LRFRP の処理系は手続きマクロと呼ばれる Rust の言語機能を利用して実装されている．これにより，LRFRP の処理系が Rust コンパイラによって実行されるようになる．DSL の処理系がホスト言語の処理系の上で動作する利点は大きい．例えば DSL を用いて開発を行う際，それ自身が Language Server や開発環境を独自に持たずとも，ホスト言語が持つそれが開発のサポートを行うことができるからである．実際，LRFRP の処理系はコンパイラとして独立したソフトウェアを持たずとも，エラーメッセージを Rust プログラムのコンパイル時に出力することができる．さらに Rust の Language Server の機能を利用できる開発環境を用いれば，このエラーメッセージを開発中に得ることができる．

Rust の標準ライブラリ std はネットワークやスレッド等の様々なホストシステムのサポートを前提にしているため，組込み開発ではしばしばこれを用いることができない．このような状況においても最低限のユーティリティを提供するために，Rust には core と呼ばれる標準ライブラリのサブセットが存在しており，LRFRP はこのライブラリのみに依存した軽量なコード生成を行うようにしている．これにより，近年 Rust の応用が進んでいる組込み開発においても LRFRP は活用可能である．

2.2 構文

LRFRP は図 1 に示す構文を持ち，これらを組み合わせて入力時変値から出力時変値への計算を表現する．提供されている機能は以下の通りである．

- プリミティブ演算・関数の使用
- ローカル時変値の宣言
- フィードバックのある計算の表現
- システムに対する初期化定数の設定

LRFRP プログラムは Rust モジュールとして変換されるため固有のモジュール名を持ち，その他入出力時変値の宣言，関数定義，時変値定義から構成されている．またリアクティブシステムで用いる定数を，Args 構造体を経由して Rust から与えることができるようになっている．時変値の定義は依存関係の宣言的な列挙であるから，その定義順序を問わない．

プログラム

P	::=	(M, D, F, V)	
M	::=	mod m;	(モジュール宣言)
D	::=	$D_d, ..., D_d$	(入出力データ定義列)
F	::=	$D_f, ..., D_f$	(関数定義列)
V	::=	$D_v, ..., D_v$	(時変値定義列)

入出力データ型定義

D_d	::=	In $\{v : \tau, ..., v : \tau\}$	(入力時変値宣言)
	|	Out $\{v : \tau, ..., v : \tau\}$	(出力時変値宣言)
	|	Args $\{v : \tau, ..., v : \tau\}$	(初期化定数宣言)

関数定義

D_f	::=	fn $f(v : \tau, ..., v : \tau)$ -> $\tau = e$;	(関数定義)

時変値定義

D_v	::=	let $v = e$;	(非 Cell 時変値定義)
	|	let $v : \tau$ <- delay e -< e;	(Cell 時変値定義)

式

e	::=	c	(定数)
	|	x	(変数参照)
	|	π	(プリミティブ演算)
	|	$f(e, ..., e)$	(関数適用)
	|	if e then e else e	(条件分岐)
	|	$\{$let $v = e$; ...let $v = e$; $e\}$	(スコープ)

図 1 LRFRP の構文

```
frp! {
  mod SimpleFanController;

  In { tmp : f32 }
  Out { fan : bool }

  let fan = tmp >= th;
  let th = 30.0 +
      if fan_delayed then -1.0 else 1.0;
  let fan_delayed: bool <- delay False -< fan;
}
```

図 2 LRFRP プログラムの例

2.3 実行モデル

LRFRP プログラムの簡単な例を図 2 に示す．

LRFRP では時変値間の依存関係を列挙していくが，時変値の宣言には二種類の構文が用意されている．

```
let fan = tmp >= th;
```

この構文で宣言される時変値 fan を **非 Cell 時変値**と呼び，fan は他の時変値 tmp と th の値に依存して計算されることを表現している．

LRFRP はプログラム上の時変値をノード，依存関係を有向辺として扱うと，循環した時変値の依存関係が無い限り，これは複数の有向非巡回グラフ (DAG) を形成する．異なる DAG 間には依存関係が存在していないことも考慮すると，これらにトポロジカルソートを行うことで，依存のない時変値から順に計算を行うような計算順序を得ることができる．入力時変値を受け取ってから全時変値の計算が終了するまでを**サイクル**と呼び，LRFRP のランタイムは 1 サイクルの間に，得られた計算順序に従った時変値の更新を行う．

仮に依存関係に循環のあるプログラムが与えられた場合，依存関係のグラフは閉路を持つため，計算順序を決定できない．この場合，LRFRP の処理系はコンパイルエラーを生成し，Rust コンパイラを経由してプログラマに伝達する．

時変値宣言のもう一つの構文は以下の通りであり，これによりフィードバックを伴う計算を可能とする．

```
let fan_delayed: bool <- delay False -< fan;
```

この構文で宣言される時変値 fan_delayed を **Cell 時変値**と呼ぶ．delay キーワードに与える定数はフィードバックにおける初期値であり，それ以降の fan_delayed の値は時変値 fan の値に応じて，1 サイクル遅れて変化する．この遅延を持つ時変値によって，時変値間の依存関係が循環していても計算順序を決定できるようになる．この詳細は 3.2.3 項で述べる．

3. 実装

LRFRP は手続きマクロ frp! として実装されている．手続きマクロは Rust の言語機能の一つであり，これは Rust プログラムのコンパイル時における自由な Rust トークンの組み替えや生成を可能にする．

図 3 は手続きマクロの単純な例である．手続きマクロ seq![*1] は連続したインデックスを用いた，コンパイル時における単純なコード生成を実現する．ここでは列挙型 Processor を宣言すると同時に，そこに含まれる Cpu0 から Cpu511 までの値の生成を行なっている．この例からわかるように，手続きマクロの呼び出しに記述されたコードは Rust の構文として有効でないトークン列であっても処理することができ，その情報を用いて新たにコード生成を行なっている．LRFRP の処理系はこの言語機能を活用しており，frp! マクロ内部に記述された LRFRP プログラムをコンパイル時に解析し，その他の Rust プログラムから利用可能なモジュールを生成する．

*1 https://github.com/dtolnay/proc-macro-workshop

```rust
use seq::seq;

seq!(N in 0..512 {
    #[derive(Copy, Clone, PartialEq, Debug)]
    pub enum Processor {
        #(
            Cpu#N,
        )*
    }
});

fn main() {
    let cpu = Processor::Cpu8;

    assert_eq!(cpu as u8, 8);
    assert_eq!(cpu, Processor::Cpu8);
}
```

図 3 手続きマクロの例：seq!

TokenStream は， Rust コンパイラが生成した AST の一部から抽出された，トークンの列を表現した構造体である．ある手続きマクロ f! の実装を行うには，この TokenStream を操作して再度 TokenStream を生成する関数 f を定義する必要がある．したがって，本研究で行なった frp! マクロの実装はすなわち frp 関数の実装である．その実装は LRFRP の構文を持つトークン列を構文解析及び意味解析し，Rust の構文を持つトークン列を生成する 3 段階の処理からなる．この変換の概略を図 4 に示す．

以降，各処理について詳細に述べていく．

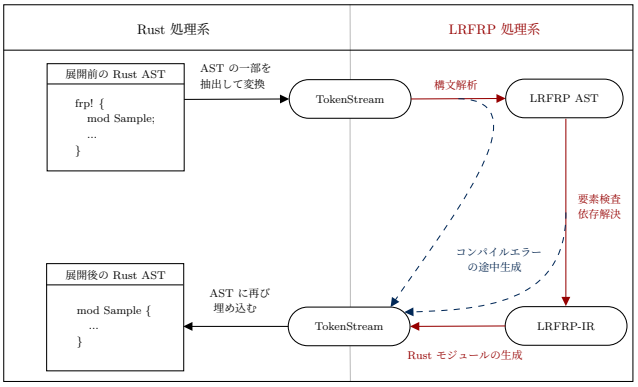

図 4 LRFRP のコンパイルの過程

3.1 構文解析

クレートとは Rust のモジュールシステムにおける最大の単位であり，実行可能バイナリを生成するプロジェクトもしくはライブラリの意味を持つ．ここでの TokenStream の解析には，手続きマクロにおける構文解析を目的とした syn クレート及び TokenStream の生成を容易にする quote クレートを用いた．構文解析では図 1 で定められた構文に従って TokenStream から LRFRP AST を構築する．LRFRP では Rust で用いることができるプリミティブ演算が提供されており，演算子の優先順位に注意して解析を行う必要がある．構文エラーが発生した場合には，コ

ンパイルエラーを生成するマクロ `compile_error!` を含む TokenStream を解析途中に返すことで，Rust コンパイラに対して構文エラーを発生させることができる．

3.2 意味解析

意味解析は，以下の 3 つの処理からなる．

- 環境抽出
- 依存抽出
- 時変値更新の順序決定

以降，各々について詳細に述べる．

3.2.1 環境抽出

ここでは LRFRP プログラムのうち，記述された依存関係の右辺式を除いた，任意の時変値や定数の宣言に関する情報を全て集める．そして同時に，得られた識別子情報に対してマーカーを付加する．このマーカーは 5 種類に分かれており，入出力時変値であるか，Cell 時変値であるか，初期化定数であるか，ローカル時変値であるか，関数であるかを示しており，コード生成を行う際に用いられる．

時変値の二重定義がなされていないこと，入力時変値が時変値の依存関係の左辺に出現していないこと，出力時変値が必ず計算されていることの確認も，この段階で行われる．

3.2.2 依存抽出

プログラムに記述された依存関係から，ある時変値を計算するために必要な時変値の情報を全て取り出す．具体的には，依存関係の右辺式に出現する全識別子に対して環境抽出にて集めた時変値情報と照らし合わせ，定義済みであれば依存関係として登録，未定義であればエラーを出力する．ここで，LRFRP は静的スコープを持つため，定義済みの時変値と同名の束縛が新しく作成したスコープ内で生成されている場合，その束縛への参照は時変値のものではないため，依存関係に含めない．

3.2.3 時変値更新の順序決定

プログラム中に Cell 時変値が存在しない場合は，依存関係に循環がない限り更新順序を決定できる．これは深さ優先探索 (DFS) を用いた Tarjan [11] のトポロジカルソートによって行なっており，以下の手順によって閉路の検出も可能にしている．

- 時変値に対して “未探索”，“探索中”，“探索済” の 3 状態を与えることを考え，初期値として全時変値に “未探索” を割り振る．
- ある時変値から開始した DFS の過程で，“未探索” の時変値を “探索中” に変更する．
- “探索済” の時変値に到達するか，探索領域が無くなるまで DFS を繰り返すが，“探索中” の時変値に到達した時は巡回した依存関係が存在しているとして，エラーを返す．
- “探索中” の時変値を全て “探索済” にし，別の “未探索” の時変値から再度 DFS を行う．

次に Cell 時変値が存在するプログラムを考える．Cell 時変値はフィードバックを実現するために，これが依存する時変値による計算結果を保持しておく必要がある．そして，前のサイクルで得られた時変値を用いた計算結果を現在のサイクルにおける出力とする．また，前の入力が存在しない最初のサイクルにおいては初期値を出力する．つまり，あるサイクルにおける全ての Cell 時変値の出力は，非 Cell 時変値の計算を行う時点で既に得られている．このため Cell 時変値との依存関係を非 Cell 時変値間の依存関係に関するトポロジカルソートに組み込む必要はなく，非 Cell 時変値の計算が終了してから Cell 時変値の計算を行う．

Cell 時変値間に依存関係がある場合，その計算順序に気をつける必要がある．ある Cell 時変値が依存する別の Cell 時変値が先に計算された場合，その時変値の出力が想定するサイクルではなく次のサイクルのものとなってしまうためである．つまり，Cell 時変値間の依存関係では非 Cell 時変値間と逆に，依存する他の Cell 時変値より先に計算を行う必要がある．したがって，Cell 時変値間の依存関係についてもトポロジカルソートを行い，得られた順序とは逆方向に Cell 時変値を計算する．Cell 時変値間に循環した依存関係が存在した場合，非 Cell 時変値の場合と同様に LRFRP の処理系はエラーを出力する．

まとめると，1 回のサイクルにおいて，以下の順序で計算を行うようなプログラムを生成すればよい．

(1) 非 Cell 時変値間の依存関係をトポロジカルソートすることで得た順序で，非 Cell 時変値の計算を行う．
(2) Cell 時変値間の依存関係をトポロジカルソートすることで得た順序の逆順で，Cell 時変値の計算を行う．

3.3 コード生成

ここでは LRFRP-IR から，Rust のモジュールを表現した TokenStream を生成する．コード生成の具体例に関しては，次章のケーススタディを参照されたい．

このモジュールは，以下の要素を含む．

- 入出力時変値及び初期化定数を表す構造体の宣言
- LRFRP で定義した関数
- Cell 時変値を表す構造体の宣言
- 出力時変値の計算を担う FRP 構造体
- FRP 構造体の実装 (メソッド)
  - `new` (コンストラクタ)
  - `cell_initializations` (Cell 時変値の初期化)
  - `run` (メソッド)
  - `sample` (出力時変値の取得)

モジュール名は LRFRP で記述したモジュール名が直接

用いられ，LRFRP で定義した関数は式を Rust の構文に合わせて生成される．ここで，この関数は LRFRP のロジックに用いることのみを用途としているため，モジュール外部へ公開しない．したがって，pub キーワードを付与せずコード生成を行うことに注意する．入出力時変値および初期化定数は Rust プログラムで初期化を行う必要があるため，メンバ変数に対するアクセスを許すために pub キーワードを付与する．Cell 時変値を表す構造体は外部に公開されず，FRP 構造体内部で状態として保持するのに用いられる．したがって，この宣言には pub キーワードは付与しない．FRP 構造体は，LRFRP が表現するシステムの状態を保持する．したがって，出力時変値及び初期化定数，そして Cell 時変値を含む型を生成する．このとき，サイクルが一度も走っていない場合に出力時変値を得ることを防ぐため，少なくとも一度サイクルを実行したかを持つフラグ running を持たせる．最後に，FRP 構造体の実装のコード生成について述べる．new メソッドは FRP 構造体のコンストラクタであり，前述の running を初期化し，その他の時変値も初期化する．現在の LRFRP ではプリミティブ型のみが使用できるため，これらの直積によって生成される型が Default トレイトを実装できることを利用し，デフォルト値で初期化する．そして最後に cell_initializations メソッドによって全ての Cell 時変値を初期化する．cell_initializations メソッドは，LRFRP プログラムで宣言された Cell 時変値を，delay キーワードの直後にある値で初期化するコードが記述される．sample メソッドは，running の値に応じて，FRP 構造体自身が持つ出力時変値への参照を Option 型に包んで出力する．ただし Option 型は std ではなく core によって提供されるものを用いていることに注意する．run メソッドは，入力時変値を受け取り，1 サイクルの計算を行う．すでに意味解析において時変値の計算順序は決定されているため，これに従って FRP 構造体が持つ状態を更新するコードを生成する．

その他のコード生成の構成要素としては，型情報と識別子がある．LRFRP の処理系は型検査を行わず Rust コンパイラに委任するため，型は直接生成コードに埋め込まれる．そのため，仮に存在しない型を LRFRP プログラムに記述した場合，LRFRP プログラムのコード生成は成功するものの，その後の Rust コンパイラによる型検査でエラーメッセージを生成される．次に識別子のコード生成であるが，LRFRP で宣言された関数や時変値の識別子の中には，改変する必要があるものが存在する．なぜなら，生成されるプログラムには各時変値に所有者が与えられるからである．例えば，ある入力時変値 v は run メソッドにおける &In 型の引数として与えられるため，これを input とすると input.v としてアクセスしなければならない．この問題に対処するために，意味解析において作成した，識別子に対するマーカーを利用する．これにより生成されるモジュールにおける時変値の所有者がわかり，適切に値を取り出すことができる．識別子が v であるような時変値や関数に対するメンバアクセス情報の生成は，以下の通りである．

- 入出力時変値の場合，input.v および self.output.v を生成する．
- Cell 時変値の場合，self.cell.v を生成する．
- 初期化定数の場合，self.args.v を生成する．
- ローカル時変値および関数の場合，何も付与しない．

4. ケーススタディ

温度および湿度のセンサーの値に基づいて換気扇のスイッチの on/off を計算するプログラム FanController を考える．模式図は図 5 の通り．

このプログラムはヒステリシス閾値を用いており，各センサーから計算された不快指数が特定の値付近で上下した際にスイッチの on/off が高速で振動しないようになっている．このリアクティブシステムはヒステリシス閾値の計算にフィードバックを含んでいるが，これを直接 LRFRP に変換することはできない．なぜなら非 Cell 時変値同士では循環した依存関係を記述できないからである．th の計算には，fan の値を 1 サイクル遅延させた Cell 時変値 fan_delayed を定義し，これをフィードバックに用いるようなプログラムを作成する必要がある．

FanController を実装した LRFRP プログラムは図 6 のようになっている．LRFRP の処理系は宣言的に記述された時変値間の依存関係を静的に解決し，その情報をもとに計算順序を決定する．これにより，時変値の宣言には 12 行目のような時変値 th の後方参照が可能になっており，フィードバックを伴う計算が簡潔に記述できるようになっている．

4.1 実行手順

frp! マクロによって生成されるモジュールは，他の Rust プログラムからは以下の API が提供されたものに見える．

- In, Out, Args 構造体および各メンバ
- FRP 構造体 (メンバアクセスは禁止)
  - コンストラクタ
  - sample/run メソッド

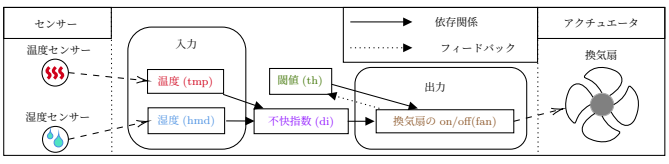

図 5 FanController の模式図

```
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() {
  let args = FanController::Args {
    fan_init: false
  };
  let mut frp = FanController::FRP::new(args);

  let mut input = FanController::In {
    tmp: 30.0,
    hmd: 60.0,
  };
  let (mut dt, mut dh) = (0.5, 1.0);

  loop {
    // 温度および湿度の変化をシミュレート
    if input.tmp > 35.0 || input.tmp < 20.0 {
      dt = -dt;
    }
    if input.hmd > 80.0 || input.hmd < 50.0 {
      dh = -dh;
    }

    input.tmp += dt;
    input.hmd += dh;

    frp.run(&input);
    let output = frp.sample().unwrap();

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

図 6　FanController を実装した LRFRP プログラム (詳細は省略)

以上の構造体およびメソッドを用いて，各実行サイクルの計算は図 8 のように行う．FRP プログラムの実行環境を提供する代わりに run メソッドと sample メソッドを設けているのは，Rust の FRP ライブラリの一つである Carboxyl [1] の API を参考にしている．

4.2　他言語との構文比較

Yampa [3] は Haskell で実装された FRP ライブラリである．このライブラリにおいて時変値は Signal と呼ばれ，Signal から別の Signal へと変換する関数は Signal Function と呼ばれる．Yampa において Signal Function は Arrow[*2] と呼ばれる構造を持ち，Arrow のコンビネータでこれらを組み合わせることで，リアクティブシステムを記述していく．Haskell のデファクトスタンダードなコンパイラである GHC の言語拡張には Arrow Syntax と呼ばれるものがあり，これは Arrow を用いた計算をより単純な構文で記述することを可能にする．図 9 は Arrow syntax

[*2] https://www.haskell.org/arrows/

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

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

  struct Cell { fan_delayed: bool }

  pub struct FRP {
    running: false,

    output: Output,
    args: Args,
    cell: Cell,
  }

  impl FRP {
    pub fn new(args: Args) -> Self {
      FRP {
        running: false,
        output: Out::default(),
        args,
        cell: Cell::default(),
      }.cell_initializations()
    }
    fn cell_initializations(mut self) -> Self {
      self.cell.fan_delayed = self.args.fan_init;
      self
    }

    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 th = 75.0 +
        if self.cell.fan_delayed {
            -0.5
        } else {
            0.5
        };
      let di = calc_di(input.tmp, input.hmd);
      self.output.fan = di >= th;
      self.cell.fan_delayed = self.output.fan;
    }
  }
}

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

図 7　変換後の LRFRP プログラム (Attribute は省略)

(1) new メソッドによって FRP 構造体のインスタンスを作成
(2) In 構造体のインスタンスに入力時変値を設定
(3) run メソッドに In 構造体のインスタンスへの参照を適用して実行
(4) sample メソッドによって Out 構造体への参照を取得
(5) Out 構造体を用いて LRFRP プログラムの出力に関する処理を行う
(6) 2. へ戻る

図 8　LRFRP プログラムの実行手順

```
fanController :: Bool -> SF (Double, Double) Bool
fanController fanInit = proc (tmp, hmd) -> do
  rec
    let di = 0.81 * tmp + 0.01 * hmd *
             (0.99 * tmp - 14.3) + 46.3
    let fan = di >= th
    let th = 75.0 + if fan' then -0.5 else 0.5
    fan' <- iPre fanInit -< fan
  returnA -< fan
```

図 9 Yampa を用いた FanController の実装例

を用いた FanController の実装である．iPre は Signal を遅延させる関数であり，これは LRFRP における delay に相当する．LRFRP における delay キーワードを用いた時変値の遅延表現が Yampa における Signal Function の適用の構文と似ていることからも分かる通り，LRFRP の構文は Arrow Syntax の影響を強く受けている．

4.3 実験

ここでは，LRFRP と Emfrp [9] の実行速度の計測およびバイナリサイズの比較を行なった．計測するプログラムは，各々の言語で FanController を実装したものである．ただし LRFRP を呼び出す main 関数は，標準ライブラリ std を利用しない組込み環境を想定したプログラムである．これをサイクル数を変えて 5050 回時間計測を行い，経過時間の平均を求めた．また，同じプログラム及びオプションでバイナリサイズも取得したところ，表 1 のようになった．

LRFRP の実行速度は Emfrp と比較して十分高速であり，バイナリサイズも小さくなっている．この理由として，LRFRP から生成される時変値更新の計算が宣言された時変値の数に対して最低限であることが挙げられる．また Emfrp のランタイムは，全ての時変値に対して現在と 1 サイクル前の値を保持しているため，それらの更新を行う計算が必要になっているという点も要因の一つであると考えられる．

表 1 LRFRP と Emfrp の比較

		LRFRP	Emfrp
OS		ArchLinux (5.6.10-arch1-1)	
CPU		Intel i7-7700HQ 3.800GHz	
ホスト言語のコンパイラ		rustc 1.45.0-nightly (fa51f810e 2020-04-29)	gcc 9.3.0
最適化オプション		-Copt-level=s -Clto=true -Cpanic=abort	-flto -Os
n サイクルの計算にかかる時間 (ms)	$n=10^6$	5.9141	24.305
	$n=3*10^6$	17.750	72.835
size コマンド	text	1,738	2,903
	data	528	624
	bss	8	96
	合計	2,274	3,623

5. 関連研究

LRFRP の他にも，Rust には既に FRP ライブラリがいくつか存在している．Carboxyl [1] は FRP を提供するクレートの一つであり，時変値を表現する構造体 Signal と，離散的に発生するイベントを表現する構造体 Stream を軸とした API が提供されている．このクレートは，すでに多くの言語で実装されている FRP ライブラリ Sodium [2] から強く影響を受けており，Signal および Stream のメソッドには Sodium が持つプリミティブが含まれている．その中でも LRFRP には存在しない機能を提供するのが switch メソッドであり，これは時間的に変化する時変値をサポートする．時間によって計算する時変値が変わるということは，時変値間の依存関係を実行時に変更できることを意味する．LRFRP では依存関係がコンパイル時に決定され，実行時には一切変化しないが，その動的な変更が可能になることにより，ゲームのロジックのような複雑なリアクティブシステムを簡潔に記述することができる．

ところで，FRP を提供する既存のライブラリの多くは豊富な計算資源を前提として作成されているため，組込みシステムの開発に直接用いることは難しい．次にこの問題の解決を試みた 3 つの関連研究を取り上げる．

Juniper [6] は Arduino をターゲットとした組込み向け FRP 言語であり，C++ で記述されたコードを生成する．Juniper は自動でのメモリ管理，高階の時変値，パラメータ多相を持つ関数といった数々の機能を提供しており，柔軟性の高いリアクティブシステムの記述が可能である．LRFRP のメモリ管理はこれを利用する Rust プログラムに依存するため，選択の余地がある．しかし，LRFRP は型システムを持たず，かつ軽量なコード生成のために時変値間の依存関係の解析をコンパイル時に行う設計を取っているため，Juniper が持つ言語機能を提供することは容易ではない．

Hae [10] は計算資源の限られた環境での動作を目的とした， Haskell をホスト言語とする内部 DSL としての FRP 言語である．Hae は Haskell の強力な型システムやモジュールの機能を利用することができ，コンパイル時にのみ存在するプリミティブ型やデータ型とそのコンビネータを組み合わせて時変値の依存関係を記述する．そしてこの依存関係を Hae のコンパイラが解析し，組み込み環境もターゲットにできる C++ のプログラムを生成する．ここで，I/O を行う箇所は Hae のコンパイラが生成した C++ 内に記述する必要があるため，FRP プログラムの実行環境を記述する際にはホスト言語の型システムやモジュールの機能を用いることができない．一方で LRFRP は Rust の言語機能であるマクロの内部に記述し，これがコンパイル時に検査されて Rust モジュールへと変換される．そして

I/O を記述した他の Rust プログラムからこれを呼び出すことで実行可能バイナリを生成する．したがって，時変値計算および実行環境を単一のプログラム上に記述することが可能であり，ホスト言語が持つ型システムやメモリ安全性に関する静的解析の恩恵を受けることができる．

Emfrp [9] は小規模組込み向けの外部 DSL としての FRP 言語であり，再帰的なデータ構造や関数の制限など，その言語機能にいくつかの制約を持たせることにより，実行時のメモリ使用量および生成されるバイナリのサイズの推定，停止性の保証を可能にしている．LRFRP はこれらの推定や保証は提供しないため，今後の課題である．Emfrp は独自に型システムを持つが，その特徴として，プリミティブ型による演算を時変値型上の計算に持ち上げる (lifting) 処理を暗黙に行う．これにより本来必要とされる明示的な型変換によってプログラムが煩雑になるのを防ぎ，時変値型とプリミティブ型を混合した演算を簡潔に記述できるようになっている．LRFRP は独自には型システムを持たず，Rust のコンパイラに型検査を委任しているが，時変値とプリミティブ値が混合した計算の記述が可能になっている点は，Emfrp の影響を強く受けている．また Emfrp は処理系は Ruby で実装されており，RubyGems を用いて個別にインストールを行う必要があるのに対し，LRFRP は言語処理系自体がマクロとして実装されているため，クレートとして容易に利用することができる．さらに，LRFRP の処理系は構文エラーや時変値の循環参照に関するエラーを Rust の Language Server を通じて通知する．そのため，独自に Language Server のサポートを開発環境に求める必要が無いなど，容易な導入を可能としている．

6. 結論と今後の課題

本研究では FRP 言語 LRFRP の処理系を実装し，ホスト言語である Rust と協調して動作することを示した．また LRFRP が標準ライブラリに依存しないコード生成を行い，組込み開発にも用いることができる FRP 言語であることを，LRFRP プログラムの例によって確認した．

LRFRP は Rust の手続きマクロとして実装されているが，手続きマクロを処理する際には他のマクロ呼び出しの持つ情報を共有することができないため，モジュール機能のように分割して記述する機能が存在しない．そのため，プログラムの規模が増加し，数百，数千行の記述が必要になった場合であっても，単一のマクロ内部に記述する必要があり，記述性の問題が発生する．また機能単位の分割による再利用性も低いため，この問題の解決は重要な課題である．

謝辞 本研究は JSPS 科研費 18K11236 の助成を受けている．

参考文献

[1] edibopp/carboxyl, `https://github.com/edibopp/carboxyl`.
[2] SodiumFRP/sodium, `https://github.com/SodiumFRP/sodium`.
[3] Courtney, A., Nilsson, H. and Peterson, J.: The Yampa Arcade, *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, New York, NY, USA, Association for Computing Machinery, p. 7–18 (online), doi:10.1145/871895.871897 (2003).
[4] Czaplicki, E. and Chong, S.: Asynchronous Functional Reactive Programming for GUIs, *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, ACM, pp. 411–422 (online), doi:10.1145/2499370.2462161 (2013).
[5] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, ACM, pp. 263–273 (online), doi:10.1145/258949.258973 (1997).
[6] Helbling, C. and Guyer, S. Z.: Juniper: A Functional Reactive Programming Language for the Arduino, *4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*, ACM, pp. 8–16 (online), doi:10.1145/2975980.2975982 (2016).
[7] Mainland, G., Morrisett, G. and Welsh, M.: Flask: Staged Functional Programming for Sensor Networks, *13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, ACM, pp. 335–346 (online), doi:10.1145/1411204.1411251 (2008).
[8] Peterson, J., Hudak, P. and Elliott, C.: Lambda in Motion: Controlling Robots with Haskell, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, Berlin, Heidelberg, Springer-Verlag, pp. 91—105 (1999).
[9] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*, ACM, pp. 36–44 (online), doi:10.1145/2892664.2892670 (2016).
[10] Sheng, W. and Watanabe, T.: Functional Reactive EDSL with Asynchronous Execution for Resource-Constrained Embedded Systems, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Studies in Computational Intelligence, Vol. 850, Springer, pp. 171–190 (online), doi:10.1007/978-3-030-26428-4_12 (2019).
[11] Tarjan, R. E.: Edge-disjoint spanning trees and depth-first search, *Acta Informatica*, Vol. 6, No. 2, pp. 171–185 (online), doi:10.1007/BF00268499 (1976).