小規模組み込みシステム向けFRP言語のための自己反映機構

渡部 卓雄[1,a)]

概要：マイクロコントローラ等の小規模組み込みシステム向けに設計された関数リアクティブプログラミング (FRP) 言語のための自己反映計算機構を提案する．提案方式では，時変値を介した実行系内部へのアクセスを可能にすることで，自己反映計算もリアクティブな操作として実現されている．本機構の導入により，小規模システムでの実行を考慮して静的に実現されている言語の実行系に，ある程度の柔軟性と適応性を与えることが可能になる．

キーワード：関数リアクティブプログラミング, 組み込みシステム, 自己反映計算

A Reflection Mechanism for an FRP Language for Small-Scale Embedded Systems

Takuo Watanabe[1,a)]

Abstract: This paper presents a simple reflection mechanism for a functional reactive programming language designed for resource-constrained embedded systems. The proposed mechanism allows accessing the internal structure of the runtime system of the language via time-varying values. This means that reflective operations are also reactive. The mechanism introduces a certain degree of flexibility and adaptability to the statically designed runtime system of the language.

Keywords: Functional Reactive Programming, Embedded Systems, Reflection

1. はじめに

リアクティブシステムとは，離散的なイベントや連続的な外界の状態変化などの入力に対して，応答の生成および自身の状態の更新をし続けるシステムである．GUI や組み込みシステムはリアクティブシステムの典型例である．

一般にリアクティブシステムに対する入力は，与えられる順序およびタイミングはあらかじめ予測できないものとされる．このことに対処するため，リアクティブシステムのプログラミングでは，コールバック，イベントループ，ポーリング等が用いられる．これらは一般的な手続き型プログラミング言語においても実現可能である一方，コードの細分化を招き可読性低下の原因となる．

リアクティブプログラミング *(Reactive Programming)* は，**時変値** *(time-varying value)* と呼ばれる抽象化機構を用いて入力およびそれらに依存する値を表現することで，リアクティブシステムの効果的な記述を支援するプログラミングパラダイムである [1]．また**関数リアクティブプログラミング** *(Functional Reactive Programming, FRP)* は，関数プログラミングに時変値を導入することでリアクティブシステムの宣言的な記述を可能にする．

時変値は時間と共に変化する値を抽象化したものである．歴史的には，時変値の概念は Haskell の対話型アニメーションライブラリ Fran[4] において導入された．そしてこれを皮切りに，例えば [9] や [6] 等，FRP の研究は主に Haskell 上のライブラリ（あるいは内部 DSL）の研究開発に関連し

[1] 東京工業大学
[a)] takuo@acm.org

ⓒ 2017 Information Processing Society of Japan

て行われてきた．これらにおいては，時変値を Haskell のデータとしてナイーブに表現した場合に発生する時間漏洩 (time leak) や空間漏洩 (space leak) といった現象を回避する必要があり，そのためにアロー (arrow) と呼ばれるデータ型 [7] の導入等が行われてきた．

アローにもとづく FRP (Arrowized FRP) では，時変値自体は隠蔽し，その代わりに時変値から時変値への関数（シグナル関数）をアロー型のデータとして導入する．そしてシグナル関数を合成するコンビネータ群を用いることで，時間・空間漏洩を回避しつつ時変値間の依存関係の記述を可能にする．

時変値を（ユーザ定義型ではなく）組み込みの型として導入することで，アローにもとづく FRP と同等の記述力を持ち，かつ簡潔な記法を持つ言語を実現できる．その一例として GUI 記述のための言語 Elm[3] がある*1．Elm では型 t の値をもつ時変値は型 $\mathsf{Signal}\ t$ を持つ．これは組み込み型であり，引数の型 t として時変値および時変値を含むデータ型は用いることはできない．時変値上の操作は関数

$$\begin{aligned}&\mathsf{lift}_n :: (a_1 \to a_2 \to \cdots \to a_n \to b) \to \\ &\quad \mathsf{Signal}\ a_1 \to \mathsf{Signal}\ a_2 \to \cdots \to \mathsf{Signal}\ a_n \to \mathsf{Signal}\ b\end{aligned}$$

によって定義できる．加えて，関数

$$\mathsf{foldp} :: (a_1 \to a_2 \to a_2) \to a_2 \to \mathsf{Signal}\ a_1 \to \mathsf{Signal}\ a_2$$

を用いることで時変値の直前の値を参照することができ，これによって状態の表現を可能にしている．

FRP の初期の研究においてロボットへの応用 [6], [9] がなされたように，FRP は組み込みシステムの記述にも適している．そして現在までに，特にマイクロコントローラ等で実現される小規模組み込みシステムのための FRP 言語として E-FRP[12], Flask[8], Céu[10], Juniper[5] などが提案されている．

著者らは，標準的な C をターゲットとした純粋関数型言語である Emfrp を設計・実装し，いくつかの例題をとおしてその有用性を明らかにしている [11]．Emfrp では時変値を一級データとして扱わず，必ず名前によって参照するというプログラミング上の制約を設けている．また一般的な再帰呼び出しおよび再帰的なデータ構造の利用を禁止する代わりに，任意の時変値の直前値を参照できる言語機構を導入している．以上によってプログラムが利用する記憶領域の大きさを静的に確定できるようになっている．また実行系としてシンプルな push 型を採用しているため，生成されるコードサイズも小さい．

このように Emfrp ではプログラムの表現力を大幅に落とすことなく小規模組み込みシステムに適した言語機構を提供している．しかしその一方でプログラムの実行方式に関する柔軟性が不足しており，例えば pull 型の実行方式を採用すれば避けられるような不要な計算を行ってしまう場合がある．

このような問題は push 型と pull 型の実行を混在させることで解決できるが，そうすることで実行系が複雑になりサイズが肥大化するという問題が生じる．筆者らは，簡単な自己反映計算機構を実行系に導入することを提案し，この問題を解決可能であることを示した [13]．現時点でこの提案の実装は行われていない．本稿では，実装にむけたアイディアについて概略を述べる．

*1 バージョン 0.17 以降では時変値の利用をやめている [2]．

2. Emfrp 概要

2.1 ノード

時変値の抽象化方式は FRP 言語の設計における主要なトピックの一つである．Emfrp における時変値は**ノード** *(node)* と呼ばれ，以下の構文で定義される．

`node` n = e　あるいは　`node init`[c] n = e

ここで n はノードの名前を表す識別子，e はノードの定義式，c は初期値である．

ノードは必ず定義された名前で参照され，ノードそのものを変数に代入したり，関数の引数にすることはできない．つまり Emfrp におけるノードは一級データではない．またノードの型は組み込み型で，ノードの値となりうる型は整数や浮動小数点数のようなスカラー型に限定される．

一般に FRP 言語のプログラムは，時変値を頂点，時変値間の依存関係を辺とする有向非循環グラフ (DAG) として表現される．Emfrp の場合，プログラムを表す DAG はコンパイル時に決定され，実行時に変化することはない．つまり実行時におけるノードの生成・削除，および依存関係の変更はできない．また，ノードの値となり得る型のサイズは静的に決まり，各ノードの値の計算（上記の式 e の評価）において繰り返しや再帰の利用は許されていない．したがって，Emfrp ではプログラムとそれが必要とする記憶領域の大きさはコンパイル時に決定される．

2.2 例題：環境センサによるファン制御器

Emfrp[11] はマイクロコントローラ等の小規模組み込みシステム向けに設計された関数リアクティブプログラミング言語である．本節では同言語の設計と実装の概略について例を通して説明する．

Emfrp ではプログラムはモジュールと呼ばれる単位で構成される．図 1 は温度・湿度センサーによるファンの制御器のためのモジュールの例である．モジュールを構成するノードには入力ノード，出力ノード，内部ノードの 3 種類がある．入力および出力ノードはモジュール定義時に予約語 `in` および `out` によってそれぞれ宣言される．図 1 では `tmp` および `hmd` が入力ノード，`fan` が出力ノードである．これらはそれぞれ温度・湿度センサおよびファンの制御回路といった外部機器に接続されている．2 つの入力ノードは

ⓒ 2017 Information Processing Society of Japan

```
1  module FanController  # module name
2  in  tmp : Float,      # temperature sensor
3      hmd : Float       # humidity sensor
4  out fan : Bool        # fan switch
5  use Std               # standard library
6
7  # discomfort (temperature-humidity) index
8  node di =
9      0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
10
11 # fan switch
12 node init[False] fan = di >= 75.0 + ho
13
14 # hysteresis offset
15 node ho = if fan@last then -0.5 else 0.5
```

図 1 環境センサによるファン制御器のためのモジュール

温度・湿度センサの現在値を表しており，出力ノードの現在値によってファンの ON/OFF が行われる．

内部ノードは外部機器との接続がないノードである．このモジュールには di および ho という内部ノードが定義されており，それぞれ不快指数とヒステリシス制御用のオフセット値（後述）を表している．センサー値を表すノード tmp および hmd の値が時間とともに変化するにしたがい，di および fan の値も変化する．結果として，温度・湿度の変化に応じてファンの ON/OFF が自動的に切り替わることになる．

2.3 履歴に依存する値の表現

図 1 の 12 行目が以下のように定義されているとする．

```
node fan = di >= 75.0
```

この場合，di の値が 75 以上であるときおよびそのときに限り fan の値が真になる．ファン制御器の動作としてはこれで良さそうである．しかしこの場合，もし di の値が 75 前後で変化するとファンの ON/OFF が頻繁に行われることになり，結果としてファン（あるいはリレー等）の故障につながる可能性が高い．

このような動作を避けるために，図 1 では簡単なヒステリシス制御を導入している．具体的には，内部ノード ho を導入し，ファンが OFF の状態から ON になるには di の値が 75.5 以上に，ON の状態から OFF になるには場合は 74.5 未満になる必要があるようにしている．

内部ノード ho の定義（15 行目）において fan@last という記法を用いている．これはノード fan の直前の値の参照を表している．

これは Elm における foldp に相当する機構であるが，foldp では各ノードがそれ自身の直前値のみを参照できるのに対し，Emfrp では@last により任意のノードの直前値を参照でき，プログラムを簡潔にできる．

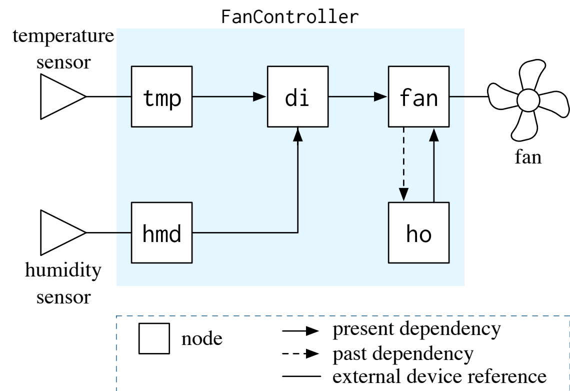

図 2 図 1 のグラフ表現

2.4 実行モデル

第 2.1 節で述べたように，Emfrp のプログラム（モジュール）は DAG として表現できる．任意の有向グラフではなく DAG としている理由は依存関係の解決のためである．図 2 は図 1 のグラフ表現である．ノード間の矢印付きの実線はノードの依存関係，つまりノードの現在値を計算するために他のノードの現在値が必要であることを示している．例えば di の現在値の計算には tmp と hmd の現在値が必要である．

一方 fan から ho へのは矢印付きの点線が引かれている．これは ho の現在値を計算するために fan の直前値が必要であることを表している．Emfrp では，点線を取り除いたグラフが DAG になっていればよい．

Emfrp の実行系は，ノードの値を順番に更新してそれぞれの現在値を計算する．このときの順番はプログラムの DAG 表現で表される半順序関係と矛盾しなければよい．つまり DAG をトポロジカルソートしたリストの順に更新を行なう．図 2 の場合，例えば tmp, hmd, di, ho, fan の順で更新を行えばよい．ここでプログラムを構成するノード全ての更新を**サイクル**と呼ぶ．Emfrp の実行はサイクルの繰り返しとみなすことができる．

各ノードの値が更新されると，そのノードは更新前の値と更新後の値をもつことになる更新後の値が現在値，更新前の値が直前値である．サイクルの中でまた更新されていないノードは直前値のみをもつことになる．

2.5 実装

Emfrp コンパイラ[*2]は Emfrp モジュールを C のコードに変換する．図 3 は図 1 からコンパイラが生成した C コードの簡略版である．変数 node_n および last_n はノード n の現在地および直前値をそれぞれ格納するために用意されている．

[*2] https://github.com/sawaken/emfrp/

```
// current and previous node values
float node_tmp, node_hmd, node_di, node_ho,
      last_tmp, last_hmd, last_di, last_ho;
bool node_fan, last_fan;

// performs a single iteration
void update() {
    // reads input node values
    Input(&node_tmp, &node_hmd);
    // updates internal/output nodes
    update_di(&node_di, node_tmp, node_hmd);
    update_ho(&node_ho, last_fan);
    update_fan(&node_fan, node_di, node_ho);
    // performs output
    Output(&node_fan);
    // records node values for next iteration
    last_tmp = node_tmp; last_hmd = node_hmd;
    last_ho = node_ho; last_di = node_di;
    last_fan = node_fan;
}

// computes the new value of di
void update_di(float *node_di, float node_tmp,
               float node_hmd) {
    *node_di = 0.81 * node_tmp + 0.01 * node_hmd
               * (0.99 * node_tmp - 14.3) + 46.3;
}

... other update functions ...

int main() {
    ... initialization code ...
    while (true) update();
}
```

図 3 コンパイラが生成する実行系（簡略版）

```
// reads the sensor values
void Input(float *tmp, float *hmd)  {
    *tmp = (float)i2c_read(ADDR_TMP);
    *hmd = (float)i2c_read(ADDR_HMD);
}

// turns the fan ON/OFF
void Output(bool *fan) {
    if (*fan) gpio_set_bit(REG_GPIO_A, PIN_FAN_SW);
    else gpio_clear_bit(REG_GPIO_A, PIN_FAN_SW);
}
```

図 4 入出力関数

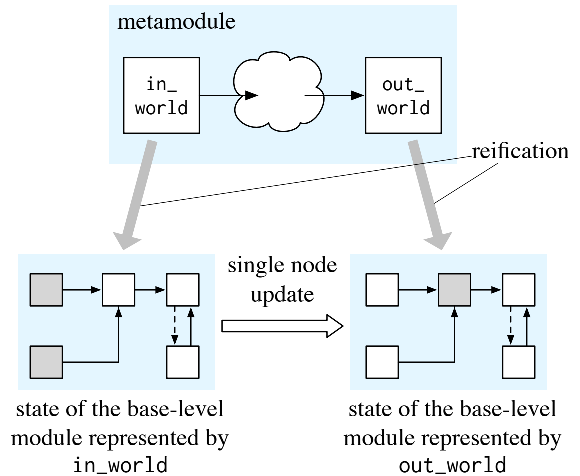

図 5 メタモジュール

関数 update は 1 回のサイクルを実行する．この関数は前節で述べたように DAG で定義される半順序と矛盾しない順序でノードの現在値の更新を行なう．個々のノードの更新は関数 update_n が担当する．

関数 Input および Output はそれぞれ入力・出力ノードと外部機器との I/O を行なう．Emfrp コンパイラはこの 2 つの関数のスケルトン（本体が空の関数定義）を生成するので，実際の入出力はスケルトンの本体をうめる形で実装する．図 4 は図 1 のための入出力関数の例である．

3. 自己反映機構

Emfrp は組み込みシステム向けに FRP の考え方にもとづく高いレベルの抽象化機構を提供している．一方，小規模システム向けを考慮した実行系を実現するため，プログラムの DAG 表現やノードの取りうる値などが静的に定義されており，そのため実行系に柔軟性や（動的）適応性が欠けている．本節では自己反映機構を導入することで実行系をカスタマイズする手法について説明する．

3.1 メタモジュール

実行系に対するアクセスを提供するためにメタモジュールの概念を導入する．これは他の自己反映言語におけるメタオブジェクトやメタクラス等に相当する．メタモジュールはアプリケーションレベルのモジュール（ベースレベルモジュール）の実行をモデル化している（図 5）．

メタモジュールの 1 サイクルは，おおむねベースレベルモジュールにおける 1 ノードの更新に相当する．したがってベースレベルモジュールの 1 サイクルはメタモジュールでは数サイクルで表現される．メタモジュールは，ベースレベルモジュールを構成するノード群を表現するためのノード in_world および out_world を持っている．これら 2 つのノードはそれぞれ（1 ノードの）更新前と更新後のベースレベルモジュールを具現化 (reify) している（図 5）．

ⓒ 2017 Information Processing Society of Japan

```
module VanillaMeta
in  in_world  : World
out out_world : World
use Reflect

node out_world = in_world of
    # Finishes a single base-level iteration
    ([], ys) -> (ys, [])
    # Updates a base-level node
    (xs@((n, p, c, e):xs'), ys) ->
        eval(e, xs, ys) of
          # Updates the current value of the node
          Just(v) -> (xs', ys ++ [(n, c, v, e)])
          # Does not update if evaluation fails
          Nothing -> (xs', ys ++ [(n, p, c, e)])
```

図 6 デフォルトのメタモジュール

メタモジュールの動作は図 6 に示す疑似コードによって表すことができる．ここでは説明のために Haskell 風のリスト記法（内包表記を含む）を用いているが，Emfrp ではリストは使うことはできない．

また e `of` $\{p_i$ `->` $e_i\}^+$ はパターンマッチ（いわゆる case 式）を表す Emfrp の記法である．Haskell と同様に，$v@p$ のようなパターンは，パターン p と同じものにマッチするが，このときマッチした式全体が変数 v にバインドされる．

ノード `in_world` および `out_world` の型 `World` は以下のように定義できる．

```
type World = ([Node], [Node])
```

これはノードを具現化したデータの型である `Node` のリストのペアを表している．このペアは，それぞれ未更新のノードの列と更新済みのノードの列を表現している．ベースレベルモジュールの 1 サイクルは，`(xs, [])` という形のノード `in_world` で開始し，`([], ys)` という形のノード `out_world` で終了する（図 6, 8 行目）．

具現化されたノードの型 `Node` は以下のように 4 つ組として表すことができる．

```
type Node = (String, Value, Value, Expr)
```

ここで 4 つ組の各要素はノードの名前，直前値，現在値，そしてノードを定義する式である．メタモジュールの入出力コードは図 7 になっており，1 サイクルが終了すると，更新されたノードのリストの更新を再び開始するようになっている．

型 `Value` はベースレベルのデータの型を表している．このデータが表現しているデータをメタモジュール内部で扱いたいときは，以下の関数 `reify` を用いる．

```
reify[a] :: Value -> Maybe[a]
```

ここで `[]` で囲まれた部分は型パラメータである．また返値の型は以下のように定義されるオプション型である．

```
// world represents the entire program state
world_t *world;

void Input(world_t *in_world) {
    in_world = world;
}

void Output(world_t *out_world) {
    world = out_world;
}
```

図 7 図 6 のための入出力コード

```
func valueOf[a](xs : [Node], n : String) : Maybe[a] =
    [ c | (m, _, c, _) <- xs, n == m ] of
        [v] -> reify[a](v)
        _   -> Nothing
```

図 8 ノードの値を得るための補助関数の例

```
type Maybe[a] = Just(a) | Nothing
```

関数 `reify` に与えられる `Value` 型の値が，型パラメータで指定された型の値である場合は `Just(v)` が，そうでない場合は `None` となる．

ノードを更新するために式を評価する関数が `eval` である．

```
eval :: (Expr, [Node], [Node]) -> Maybe[Value]
```

第 2，第 3 引数はノードのリストである．これらはそれぞれ直前値と現在値の参照に用いられる．

3.2 時変値を介した自己反映操作

図 8 に示したような関数を使うことで，メタモジュール内部においてベースレベルモジュールのノードの値（現在値および直前値）を参照することができる．これにより，ベースレベルのノードを介して実行系（メタレベル）の動作を変えるようなプログラムを書くことが可能である．これはつまり時変値を介した自己反映操作であり，自己反映操作自体がリアクティブなものと解釈できる．

自己反映操作を含む例題の記述は [13] にある

4. まとめと今後の課題

メタモジュールの考え方を使った Emfrp 用の自己反映計算の 1 方式を提案した．この方式では，ベースレベルにおける時変値（ノード）を介した自己反映動作が可能になる．

提案方式の実装は現在行なっている．本稿の疑似コードでは型 `World` はリストを使って表現したが，Emfrp ではリストのような実行時にサイズの決まるデータは使えないため，実装ではサイズを固定したキューのようなデータ構造を用いる予定である．

参考文献

[1] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S. and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, Vol. 45, No. 4, p. 52 (online), DOI: 10.1145/2501654.2501666 (2013).

[2] Czaplicki, E.: A Farewell to FRP: Making signals unnecessary with The Elm Architecture (2016).

[3] 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).

[4] 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).

[5] 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 (FRAM 2016)*, ACM, pp. 8–16 (online), DOI: http://dx.doi.org/10.1145/2975980.2975982 (2016).

[6] Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming, *Advanced Functional Programming*, Lecture Notes in Computer Science, Vol. 2638, Springer-Verlag, pp. 159–187 (online), DOI: 10.1007/978-3-540-44833-4_6 (2003).

[7] Hughes, J.: Generalising monads to arrows, *Science of Computer Programming*, Vol. 37, No. 1–3, pp. 67–111 (online), DOI: 10.1016/S0167-6423(99)00023-4 (2000).

[8] Mainland, G., Morrisett, G. and Welsh, M.: Flask: Staged Functional Programming for Sensor Networks, *International Conference on Functional Programming (ICFP 2008)*, ACM, pp. 335–346 (online), DOI: 10.1145/1411204.1411251 (2008).

[9] Pembeci, I., Nilsson, H. and Hager, G.: Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages, *4th International Conrefernce on Principles and Practice of Declarative Programming (PPDP 2002)*, ACM, pp. 168–179 (online), DOI: 10.1145/571157.571174 (2002).

[10] Sant'Anna, F., Ierusalimschy, R. and Rodriguez, N.: Structured Synchronous Reactive Programming with Céu, *14th International Conference on Modularity (Modularity 2015)*, ACM, pp. 29–40 (online), DOI: 10.1145/2724525.2724571 (2015).

[11] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *Modularity 2016 Constrained and Reactive Objects Workshop (CROW 2016)*, ACM, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).

[12] Wan, Z., Taha, W. and Hudak, P.: Event-Driven FRP, *Practical Aspects of Declarative Langauges*, Lecture Notes in Computer Science, Vol. 2257, Springer-Verlag, pp. 155—172 (online), DOI: 10.1007/3-540-45587-6_11 (2002).

[13] Watanabe, T. and Sawada, K.: Towards Reflection in an FRP Language for Small-Scale Embedded Systems, *Programming 2017 Workshop on Live Adaptation of Software SYstems (LASSY 2017)*, ACM, (online), DOI: 10.1145/3079368.3079387 (2017).

ⓒ 2017 Information Processing Society of Japan