T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

論題(和文)	組込みシステム向けFRP言語における状態依存動作のための抽象化機構
Title(English)	An Abstraction Mechanism for Modeling Stateful Behaviors in an FRP Language for Embedded Systems
著者(和文)	松村有倫, 渡部卓雄
Authors(English)	Arimichi Matsumura, Takuo Watanabe
出典(和文)	情報処理学会論文誌（プログラミング）, Vol. 13, No. 2, pp. 1-13
Citation(English)	IPSJ Transactions on Programming (PRO), Vol. 13, No. 2, pp. 1-13
発行日 / Pub. date	2020, 4
権利情報 / Copyright	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。 Notice for the use of this material The copyright of this material is retained by the Information Processing Society of Japan (IPSJ). This material is published on this web site with the agreement of the author (s) and the IPSJ. Please be complied with Copyright Law of Japan and the Code of Ethics of the IPSJ if any users wish to reproduce, make derivative work, distribute or make available to the public any part or whole thereof.

Powered by T2R2 (Science Tokyo Research Repository)

情報処理学会論文誌　プログラミング　**Vol.13　No.2　1–13 (Apr. 2020)**

組込みシステム向けFRP言語における状態依存動作のための抽象化機構

松村 有倫[1,a)]　渡部 卓雄[1,b)]

受付日 2019年8月30日, 採録日 2019年12月17日

概要：組込みシステムは，外部からの入力に対して自身の状態を更新しながら応答を続けるリアクティブシステムの典型例である．組込みシステムの多くは状態遷移を含み，状態に応じて振舞いを変化させる必要があるためプログラムが複雑になる．本研究では，小規模組込みシステムを対象とした関数リアクティブプログラミング（FRP）言語である Emfrp に対し，システムの状態に応じて計算の動的な切替えを行うための抽象化機構を導入する switch 拡張を提案する．本拡張は状態ごとに時変値と次に遷移する状態の計算を記述するための機構を提供し，FRP の枠組みの中で状態遷移を扱うことを可能にする．これによって状態遷移を含むリアクティブシステムを宣言的に記述することができる．小規模組込みシステムの開発はマイクロコントローラなどの計算資源が限られた環境を対象としている．このような環境ではリソースの不足による実行時エラーを防ぐため静的にメモリ消費量を見積もる必要がある．提案する手法では遷移する状態の集合を静的に限定し，動的なメモリの確保を行うことなく実行時の計算の切替えを実現させている．switch 拡張の処理系は純粋な Emfrp への変換器として実装されている．提案する拡張によって，状態に応じて振舞いが動的に変化するコンポーネントの記述が容易になることを例を通して示す．

キーワード：関数リアクティブプログラミング，組込みシステム

An Abstraction Mechanism for Modeling Stateful Behaviors in an FRP Language for Embedded Systems

Arimichi Matsumura[1,a)]　Takuo Watanabe[1,b)]

Received: August 30, 2019, Accepted: December 17, 2019

Abstract: Embedded systems are classified as reactive systems, which respond to external input and update their states. Developing such systems tend to be complicated because many of those systems change their behavior depending on the internal states. To support developing embedded reactive systems, we propose an abstraction mechanism that provides a method to express state-dependent behaviors in the style of Functional Reactive Programming (FRP). Specifically, we introduce a variant of switch-extension to Emfrp, an FRP language designed for small-scale embedded system. This extension provides a mechanism to define time-varying values and state transitions for each state. It enables us to describe stateful reactive systems in a declarative way. Also, the language targets small-scale embedded systems such as microcontrollers. Thus, dynamic memory allocation is not desirable because it may cause runtime error due to unpredictable memory consumption. The proposed extension statically defines the set of states included in transitions to limit possible behaviors. We implement the switch-extension as the source-code translation to pure Emfrp codes. We show an example to demonstrate that the extension adequately modularizes stateful components.

Keywords: functional reactive programming, embedded systems

1. はじめに

リアクティブシステムは，外部からの入力に応答して内部状態の更新と出力を行い続けるシステムである．**関数リアクティブプログラミング（Functional Reactive**

[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)] arimichi@psg.c.titech.ac.jp
[b)] takuo@acm.org

© 2020 Information Processing Society of Japan

Programming, FRP）[4], [7], [12] は時間とともに変化する値の抽象化である**時変値**（**time-varying value**）を用いてリアクティブシステムの記述を支援するプログラミングパラダイムである．外部からの入力は時変値として表され，それらを組み合わせて新たな時変値を定義することによって入力に対する処理を宣言的に記述することができる．入力に対する応答は時変値の再計算という形で実現されることになる．アニメーション [4]，GUI [3]，ロボティクス [7], [12] など，FRP は様々な分野のリアクティブシステムに適用され有用性が示されてきた．ロボットに限らずほとんどの組込みシステムはリアクティブシステムに分類することができ，FRP の導入による開発の支援を見込むことができる．しかし，文献 [6], [8], [9] で提案されているものなどの例外を除き，FRP を導入した言語やライブラリの多くは豊富な計算資源を前提としており，マイクロコントローラなどの計算資源が限られた環境に直接適用することは困難である．

小規模組込みシステムにおけるリアクティブシステムの開発を支援するため，我々は FRP 言語 Emfrp [14] の設計・実装を行ってきた．小規模組込みシステムの開発においては豊富なメモリ資源や十分な CPU 性能を持たない環境（たとえば主記憶が数 KB 程度のマイクロコントローラ）がターゲットになる．このような環境でも利用可能にするため，我々は Emfrp を設計するにあたり，実行コードのメモリフットプリントを小さくすることと同時に，動的なメモリ管理を避けて使用メモリ量を静的に決定できるようにした．後者について具体的には，関数やデータ型の再帰的な定義を禁じたこと，時変値を一級データとせず必ず名前で参照させること，高階の時変値を扱えないようにしたことといった構文および型システム上の制約を設けた．また，時変値の過去の値の参照は直前値（後述）のみに制限しており，FRP の実装の際に問題となる空間漏れを解決している．

一方でこれらの制約から，Emfrp は実行時に動作を変化させる適応的な動作の記述を苦手としている．多くのリアクティブシステムは状態遷移をともなっており，遷移する状態に応じてプログラムの振舞いを変化させる必要がある．しかし，Emfrp には高階の時変値など，時変値の依存関係を動的に変化させるような言語機構は存在していない．この問題に対し，著者の 1 人は Emfrp に対する言語拡張として，適応的な動作のモジュール化を支援する文脈指向プログラミング（COP）機構の導入 [16] を提案した．この拡張では文脈（実行時情報）に依存して変化する振舞いを記述するための層と呼ばれる機構を提供している．文脈に応じて適切な層を活性化させ，時変値の定義を上書きすることによって実行時の振舞いの変化を実現させている．しかし，変化する振舞いの定義の際にはベースとなる時変値定義が必須になるため，状態遷移を管理して計算の切替えを行うような振舞いはうまくモジュール化できない場合がある．

この問題を解決するため，本研究では状態に依存して振舞いが変化するコンポーネントの記述に特化した言語機構である **switch 拡張**を提案する．本研究の貢献は，この拡張の導入によって状態依存動作を含む組込みリアクティブシステムを宣言的にモジュール化して記述することを可能にした点にある．この拡張は Haskell の FRP ライブラリである Yampa [7] における `switch` オペレータに相当する機構を Emfrp に導入するもので，時変値に応じて更新計算に用いる式を切り替えることを可能にする．

本研究で提案する switch 拡張では，アプリケーションプログラムがとりうる状態とそれらの間の遷移動作を定義するための **switch モジュール**を導入する．このモジュールでは，状態ごとに関連する出力時変値と状態遷移動作を定義する．状態遷移にともなう振舞いの変化は，出力時変値の計算を切り替えることにより実現される．Yampa における `switch` オペレータとの違いは，実行時に切り替える振舞いの集合が switch モジュール内の状態定義という形で明示されることにある．このため，提案手法ではステートマシンによる状態遷移の設計を素直に反映した形でプログラムを記述できる．これにより，入力に対する応答を宣言的に記述できる FRP の利点を享受しつつ直感的に状態遷移を扱うことができるようになる．また，システムがとりうる状態はプログラム中で定義したものに限定されるため，実行時に切り替える計算の集合をコンパイル時に決定することができる．したがって，静的にメモリの使用量を見積もることができる Emfrp の特徴を保ったまま動的な計算の切替えを実現できるようになる．

本論文は次のように構成される．2 章で Emfrp の概要について説明し，続く 3 章では例を通して状態遷移を扱う際の問題点を明らかにする．4 章では switch 拡張について述べ，3 章であげた問題点を解決していることを示す．5 章では純粋な Emfrp への変換による switch 拡張の実装について説明する．6 章で関連研究の紹介を行い，7 章で本論文のまとめと今後の課題について述べる．

2. Emfrp

Emfrp [14] は小規模組込みシステムを対象とした関数リアクティブプログラミング（FRP）言語である．本章では switch 拡張を導入する前の Emfrp（以下，純粋な Emfrp と呼ぶ）の概要と主な構文について説明する．

2.1 概要

関数リアクティブプログラミング（**Functional Reactive Programming, FRP**）は，**時変値**（**time-varying value**）と呼ばれる抽象化機構を用いることで，リアクティブシステムの動作の宣言的な記述を可能にするプログラミ

ⓒ 2020 Information Processing Society of Japan

ングパラダイムである．FRPにより，リアクティブな動作の記述において典型的なポーリングやコールバックなどを排除し，プログラムを見通し良く記述することができる．

時変値は時間とともに変化する値を抽象化したものである．リアクティブシステムにおける外界からの入力および外界への出力，および入力から出力を計算する際に用いられる値を時変値として表現し，それらの間の依存関係を式として表現したものがFRPのプログラムである．ある時変値 x を定義する際に他の時変値 y を用いているとき，いい換えると x を定義する式中に y が用いられているとき，x は y に依存するという．FRPのプログラムに含まれる時変値を接点とし，時変値 x が y に依存することを y から x への有向辺とするグラフを構成することができる．Emfrpでは時変値をノードと呼び，予約語 `node` を用いて定義する．

図 1 はEmfrpによる扇風機の制御システム（文献 [16] の例を簡略化したもの）を構成するノード（時変値）とそれらの間の依存関係を表すグラフである．ノード `tmp` と `hmd` は，それぞれ温度センサおよび湿度センサと連動しており，両センサの現在の計測値を表している．ノード `di` は不快指数を表しており，その値は `tmp` と `hmd` から計算される．この `di` を用いて定義される真偽値型のノード `fan` は扇風機のON/OFF状態を表しており，扇風機のスイッチと連動している．このシステムは温度と湿度を計測し，それらの値から計算される不快指数が閾値以上になったときに扇風機をONにする．

この例における `tmp` と `hmd` のように，センサなどの入力機器と連動し，入力の変化に応じて値が更新されるノードを**入力ノード**と呼ぶ．また，`fan` のように出力機器と連動し，その値が出力に反映されるノードを**出力ノード**と呼ぶ．

図 1 に相当するEmfrpのプログラムが**図 2** である．Emfrpのプログラムはモジュールという単位で記述される．記述の冒頭ではモジュール全体の設定を指定するヘッダを記述する（1–5 行目）．1 行目はモジュールの名前（`FanController`）を表しており，2–3 行目と 4 行目はそれぞれ入力ノード（`tmp` と `hmd`）と出力ノード（`fan`）を宣言している．5 行目はモジュールの利用するライブラリ（`Std`）を指定している．`Std` はEmfrpの提供する標準ライブラリである．**図 3** に示すように，ライブラリは**マテリアルファイル**としてユーザが定義することもできる．マテリアルファイルではデータ型，定数，関数を定義することができ，複数のモジュールからこれらを利用することを可能にする．

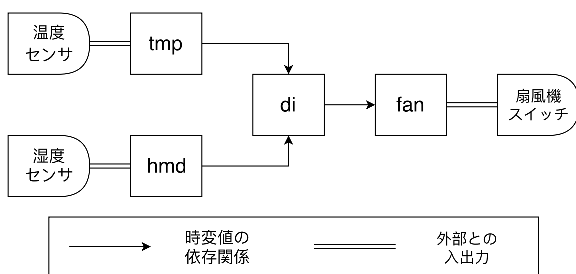

図 1　扇風機制御システム

Fig. 1　A fan controller system.

ヘッダに続いて定義部（7–15 行目）を記述する．定義部ではノードを組み合わせて新たなノードを定義する（8–9, 12 行目）．定義中で他のノードを参照することによってそのノードに依存したノードを定義できる．ただし，入力に指定されているノードは外部入力を抽象化するノードであるため定義することなく参照できる．また，モジュール内部にもデータ型，定数，関数を定義することができ（15 行目），ノードの定義に用いることができる．

2.2　フィードバックの表現

FRPを用いて記述されたリアクティブシステムの内部状態は，各時点における時変値のスナップショットと考えることができる．図 2 では，各ノードの値は入力ノードの値のみで決まり，出力は実行履歴に依存しない．実行履歴に依存する振舞い，すなわち内部状態を持つようなプログラムを表現するため，FRPをサポートする言語やライブラリの多くは時変値の直前の値を参照するフィードバックループに相当する機構を提供している．Emfrpでは `@last` というオペレータをノードに適用することで任意のノードの直前値を参照できる．

この機能を用いた例として，ボタンを押した回数を数えるノード `buttonCount` の定義を以下に示す．

```
node init[0] buttonCount = buttonCount@last +
     if !button@last && button then 1 else 0
```

```
 1 module FanController  # module name
 2 in  tmp : Double,     # temperature sensor
 3     hmd : Double      # humidity sensor
 4 out fan : Bool        # fan switch
 5 use Std
 6
 7 # discomfort (temperature-humidity) index
 8 node di = 0.81 *. tmp +.
 9     0.01 *. hmd *. (0.99 *. tmp -. 14.3) +. 46.3
10
11 # fan switch
12 node fan = di >=. th
13
14 # threshold
15 data th = 75.0
```

図 2　Emfrpによる扇風機制御プログラム

Fig. 2　Implementation of the fan controller in Emfrp.

```
1 material MyLibrary
2
3 # constant value definition
4 data foo = 42
5 data bar = foo * 2 + 4
6
7 # function definition
8 func min(a : Int, b : Int) =
9   if a < b then a else b
```

図 3　マテリアルファイルの例

Fig. 3　An example of a material file.

© 2020 Information Processing Society of Japan

ここでは真偽値ノード buttonでボタンの押下状態を表している．この値はボタンが押されている間のみ真であるとする．条件式の条件 !button@last && button は，ボタンの直前の状態が偽で現在の状態が真であること，すなわち現在ボタンが押された瞬間であることを表している．つまりこの例では，ボタンが押された瞬間に buttonCount の値をその直前値 buttonCount@last に 1 を足したものとしており，結果としてボタンが押された回数を正確に数えることができる．

オペレータ @last によって直前値が参照されるノードには初期値を指定する必要がある．プログラムが起動した時点での直前値の参照は指定した初期値となる．予約語 **node** によって定義されるノードについては，初期値は **init** を用いて指定する．上の例では **init**[0] によって buttonCount の初期値を 0 としている．入力ノードについては **node** による定義がないため，ヘッダにおける宣言中で以下のように記述することで初期値を指定できる．

```
in button(False) : Bool
```

2.3　モジュールの利用

定義したモジュールは **newnode** という記法を用いて他のモジュールからサブモジュールとして利用できる．図 **4** は MainModule から SubModule をサブモジュールとして利用している例である．SubModule の入力 x，y は MainModule における x と x+1 によって与えられ，ノード a および b をサブモジュールの出力 sum および pro それぞれに束縛している．

サブモジュールは内部の定義をその場に展開することによってサポートされている．MainModule は処理系の内部で図 **5** のように展開される．サブモジュールの入力ノードの定義には **newnode** の引数が埋め込まれ，a，b はサブモジュールの出力ノードを参照するノードとして定義されている．モジュールがネストして利用されている場合には再帰的にこのような展開が行われる．

```
1 # MainModule.mfrp
2 module MainModule
3 in x : Int
4 out a : Int, b : Int
5 use Std
6
7 newnode a, b = MySubModule(x, x + 1)
```

```
1 # MySubModule.mfrp
2 module MySubModule
3 in x : Int, y : Int
4 out sum : Int, pro : Int
5 use Std
6
7 node sum = x + y
8 node pro = x * y
```

図 **4**　サブモジュールの利用

Fig. 4　An example usage of a submodule.

2.4　実行モデル

Emfrp は入力ノードの変化を依存するノードに順次伝播させていく push 型 [2] と呼ばれる実行モデルをとっている．入力時変値の更新を行い，依存関係の順に値を再計算するというイテレーションを休みなく逐次的に繰り返すことで動作する．このようなシンプルなアルゴリズムを採用することで，OS などによるスケジューラのサポートを仮定することなく動作できるようになっている．ノードの初期値に指定がある場合，最初のイテレーションが始まる前に初期化される．

2.1 節で述べたように，Emfrp のプログラムにおけるノード間の依存関係は有向グラフとして表現できる．時変値の依存関係に循環がある場合，更新計算の順序を決定することができなくなってしまうため，依存関係のグラフは非巡回有向グラフ（DAG）である必要がある．一方で，@last によるノードの参照はイテレーションが始まる前に値が計算されているため，計算の順序に影響しない．そのため，Emfrp では依存関係グラフから @last による依存関係を除いたグラフが DAG であることを要求している．更新計算の順序はこのグラフをトポロジカルソートすることによって決定している．図 1 の例では，たとえば tmp，hmd，di，fan のような更新順序を考えることができる．

Emfrp のモジュール定義は上記のアルゴリズムを実装した C や C++のコードにコンパイルされ，これらをサポートするプラットフォームで利用することができる[*1]．ユーザは入力ノードの更新処理と出力ノードの値に対する処理を追加で実装して利用する．これによってプラットフォームごとの入出力の違いを吸収し，様々な環境で動作させることを可能にしている．

3.　モチベーション

Emfrp では時変値の依存関係がコンパイル時に決定し変化しないという言語の特性上，実行時に計算を変化させるような振舞いの記述を苦手としている．本章では，例題を交えて状態依存動作を記述する際の問題点を説明する．な

```
 1 module MyMainModule
 2 in x : Int
 3 out a, b
 4 use Std
 5
 6 # expand submodule
 7 node x_sub = x
 8 node y_sub = x + 1
 9 node sum_sub = x_sub + y_sub
10 node pro_sub = x_sub * y_sub
11
12 # refer submodule output
13 node a = sum_sub
14 node b = pro_sub
```

図 **5**　サブモジュールの展開

Fig. 5　Expansion of a submodule.

*1　https://github.com/psg-titech/emfrp

© 2020 Information Processing Society of Japan

お，本論文で扱う例題の実装はすべて公開されている[*2]．実装の詳細についてはそちらを確認されたい．

3.1　例題

例題として，3 つのボタン（A，B，C）と時・分・秒の桁を持つディスプレイを持つ時計のシステムの実装を考える．このシステムを図 **6** に示す状態遷移で設計する．Display は時刻の表示を行うモード，Set(Hour)，Set(Min)，Set(Sec) は時刻のそれぞれの桁を合わせるモードである．初期状態は Display になっており，A ボタンによって時刻合わせを行う桁の切替えを行う．時刻合わせのモードでは B ボタンによって時刻をすすめる．

このシステムを Emfrp によって実装した例が図 **7** である．モジュール `Watch` は，ボタン入力を表すノード `button`

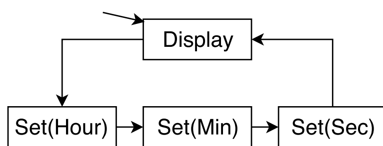

図 **6**　時計の状態遷移
Fig. 6　State transition of a digital watch.

```
 1 module Watch
 2 in button : Button,
 3    elapsed(0) : Int
 4 out display : Time
 5 use Std, Watchlib
 6
 7 type SetPos = Sec | Min | Hour
 8 type Mode = Display | Set(SetPos)
```

```
17 node curTimeUpdate : OptTime =
18   mode@last of:
19     Set(Sec) -> if buttonA(button) then
20                   SomeT(display)
21                 else NoneT
22     _ -> NoneT
23
24 node init[Time(0,0,0)] curTime =
25   curTimeUpdate of:
26     SomeT(t) -> t
27     _ -> if pulseCLK then
28            incTime(curTime@last)
29          else curTime@last
```

```
40 node init[Time(0,0,0)] display =
41   mode@last of:
42     Display -> curTime@last
43     Set(_) -> if buttonB(button) then
44                 addTime(display@last, dh, dm, ds)
45               else display@last
46
47 node init[Display] mode =
48   if buttonA(button) then
49     mode@last of
50       Display -> Set(Hour),
51       Set(Hour) -> Set(Min),
52       Set(Min) -> Set(Sec),
53       Set(Sec) -> Display
54   else mode@last
```

図 **7**　Emfrp による時計システムの実装
Fig. 7　Implementation of the digital watch in Emfrp.

[*2] https://github.com/psg-titech/EmfrpWatch

とシステムが起動してからのミリ秒単位での時間を表すノード `elapsed` を入力としてディスプレイの表示 `display` を計算する．`Watchlib` は関連するデータ型や関数をまとめたライブラリである．モジュールのモードは 7，8 行目で定義したデータ型を通じて行う．現在のモードはノード `mode`（47–54 行目）によって管理され，A ボタンが押された際，直前のモード（`mode@last`）を元に遷移先のモードを計算する．

17–29 行目は時計の時刻を管理するコードである．`curTimeUpdate` は時刻の更新イベントを表すノードである．型 `OptTime` は何らかの `Time` 型の値 `t` を持つことを表す `SomeT(t)` という値と，何も値を持たないことを表す `NoneT` という値からなる．時刻の更新イベントがあることを `SomeT(t)` で，更新イベントがないことを `NoneT` で表現している．ノード `curTimeUpdate` は時刻の桁合わせが終わるタイミング，すなわちモードが `Set(Sec)` のときに A ボタンが押されたタイミングで現在ディスプレイに表示されている時刻を更新イベントとして送出する．現在時刻はノード `curTime` によって管理される．更新イベントがある場合，このノードはイベントが保持する値に更新される．そうでない場合については 1 秒経過するごとに時刻のインクリメントを行う．図 7 では省略されているが，入力ノード `elapsed` を用いて 1 秒おきのパルスを管理するノード `pulseCLK` が定義されており，インクリメントのタイミングはこのノードで管理されている．40–45 行目は出力ノード `display` の定義である．時刻を表示するモードでは現在の時刻を表示し，時刻合わせモードでは B ボタンで時刻を進める処理を行っている．時刻合わせモードにおいて各桁で進める数値が `dh`，`dm`，`ds` というノードとして定義されている．

図 7 では `mode@last` に対する条件分岐が様々な場所に現れている．省略した部分も含めると，今回のコードでは 10 のノード定義のうち 6 つがこのような条件分岐を持っている．さらに，`dh`，`dm`，`ds` といった特定のモードにのみ影響するノードが他のノードと区別されることなく存在している．これらの要因から状態ごとの振舞いを把握することが困難になっており，可読性の低下を招いている．

3.2　例題の拡張

前節の例題を拡張してストップウォッチ機能を実装する．状態遷移を図 **8** のように拡張し，C ボタンによる時刻表示モードからストップウォッチモードへの遷移を追加する．ストップウォッチモードには停止状態と動作状態の 2 状態があり，B ボタンでスタート・ストップの操作を行う．A ボタンはリセットボタン，C ボタンは時計モードに遷移するボタンに割り当てられているが，これらの操作は停止状態でのみ行える．

行った拡張は図 **9** にまとめられる．8–9 行目では型 `Mode`

© 2020 Information Processing Society of Japan

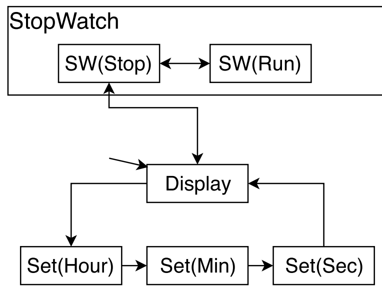

図 8　拡張後の時計の状態遷移
Fig. 8　State transition of an extended digital watch.

```
 7 type SetPos = Sec | Min | Hour
 8 type SWState = Stop | Run
 9 type Mode = Display | Set(SetPos)
10           | SW(SWState)
```

```
33 node init[0] counterSW =
34   mode@last of:
35     SW(Run) ->
36       (counterSW@last + dt) % 1000
37     _ -> 0
38
39 node pulseSW = counterSW@last > counterSW
40
41 node init[Time(0,0,0)] stopWatch =
42   mode@last of:
43     SW(Stop) -> if buttonA(button) then
44                   Time(0,0,0)
45                 else stopWatch@last
46     SW(Run) -> if pulseSW then
47                  incTime(stopWatch@last)
48                else stopWatch@last
49     _ -> Time(0,0,0)
```

```
68 node init[Display] mode =
69   if buttonA(button) then
70     mode@last of
71       Display -> Set(Hour),
72       Set(Hour) -> Set(Min),
73       Set(Min) -> Set(Sec),
74       Set(Sec) -> Display,
75       _ -> mode@last
76   else if buttonB(button) then
77     mode@last of
78       SW(Stop) -> SW(Run),
79       SW(Run) -> SW(Stop),
80       _ -> mode@last
81   else if buttonC(button) then
82     mode@last of
83       Display -> SW(Stop),
84       SW(Stop) -> Display,
85       _ -> mode@last
86   else mode@last
```

図 9　拡張した時計システムの実装
Fig. 9　Implementation of the extended digital watch.

を拡張してストップウォッチモードに対応する値を追加している．33–49 行目はストップウォッチのカウントを管理するノード stopWatch を定義している．counterSW はストップウォッチ用のパルスを管理するカウンタである．1,000 ミリ秒のカウントを行い，カウンタが 1 周したタイミングを pulseSW がとらえることで 1 秒おきのパルスを表現している．ストップウォッチの動作中以外では counterSW のカウントをリセットしておき，パルスの生成を止めている．stopWatch は停止状態における A ボタンによるリセットと動作状態におけるカウントのインクリメントを実装しており，ストップウォッチモード以外の状態ではカウント 0 を保持する定義になっている．display の定義はストップウォッチモードのときに stopWatch を表示するように修正しなければならない．mode の定義も状態遷移の拡張に合わせては 68–84 行目のように変更されているが，遷移に関係するボタンが増えたために記述が複雑になっている．

各所で条件分岐を行うことによって状態に応じた振舞いの変化を実装している場合，拡張を行う際には display や mode などの既存の定義を直接修正しなければならない．また，修正が必要ない箇所についても状態の追加によって振舞いが変化しないことを確認する必要がある．このように状態に対する条件分岐が散在するコードは拡張性・メンテナンス性の低下を招いてしまう．

時変値の定義が実行時に変化しないという制約によって，Emfrp は環境を選ばないシンプルな実行モデルで動作することができ，またメモリ消費量を一定かつ少量に抑えることに成功している．一方，状態ごとの振舞いをモジュール化して記述し，それらを実行時に切り替えられるような抽象化機構を提供していないため，上記のような問題が起こってしまう．

4.　提案手法

前章であげた状態依存動作記述のモジュール性の問題を解決するため，Emfrp に対する switch 拡張を提案する．この拡張が導入する switch モジュールでは，遷移する状態に分けて振舞いの定義を行う．本章では前章と同じ例題を通じて switch 拡張の紹介を行いつつ，前章の実装との比較を行う．

4.1　拡張の概要

新たに導入する switch モジュールではトップレベルに遷移する状態を定義する．状態の定義は，状態が管理するノードと次状態への遷移を **state** ブロックに定義することによって行う．状態にはパラメータを設けることができ，定義の内部で定数として参照できる．

各状態に対して定義されるノードは，ヘッダで指定する出力ノードと，それ以外のノードである内部ノードの 2 つに分けられる．出力ノードはモジュールの出力となるノードであり，すべての状態で必ず定義する必要がある．出力ノードはすべての状態で共有するノードとして扱われ，直前値は状態間で引き継がれる．また，初期値は入力ノードと同様にヘッダで指定される．一方，内部ノードは他の定義を補助するための状態に固有なノードである．内部ノードは定義された状態の外部から参照することはできず，直前値は状態が切り替わる際にリセットされる．なお，switch モジュールでは 5 章で紹介する実装の都合上，すべてのノードに対して初期値を要求している．また，ノー

© 2020 Information Processing Society of Japan

ドの定義はすべて状態定義の内部で行われ，状態の外部にノードを定義することはできない．

次状態への遷移は予約語 **switch** の後に記述する．すべてのノードを更新した後，switch 節の式を用いてモジュールの状態が更新される．初期状態の指定はモジュールのヘッダで行う．これを可能にするため，ヘッダには **init** 部が追加されている．

4.2　状態依存動作の記述

拡張前の例題を switch モジュールを用いて実装した例が**図 10** である．この例では，遷移する状態として時刻の表示状態 `Display` と時刻合わせ状態 `Set` を定義している．`Set` には時刻合わせを行う桁を表す `pos` がパラメータとして設けられており，時・分・秒のセット状態がまとめて定義されている．初期状態にはヘッダで `Display` を指定している．

入力ノードにはシステム起動からの時間 `elapsed` の代わりに現在時刻 `curTime` が加わっており，出力ノードには時刻の更新イベント `curTimeUpdate` が追加されている．現在時刻の管理はすべての状態に共通して行う処理であるため，この実装では関連するノードの定義を上位のモジュールに移している．入出力ノードの変更はこれにともなうものであり，現在時刻の参照や更新要求などはモジュールの入出力を通して行うように変更されている．

宣言した各状態では出力ノード `display`，`curTimeUpdate` の定義を行っている．`Set` 状態の定義では B ボタンによる桁のインクリメント処理と時刻の更新イベントの送出を実装している．`display` の定義中にある **`Retain`** は switch 拡張で導入したキーワードである．これは自身の直前値を表すキーワードであり，ここでは `display@last` を指している．ノード `dh`，`dm`，`ds` は `Set` 状態の内部ノードとして定義されている．状態遷移の定義では A ボタンによる桁の移動を実装しており，現在の状態の保持は **`Retain`** を用いて記述されている．

提案する switch 拡張を用いた実装では各状態の振舞いが切り離されて記述されており，状態に対する条件分岐が少なくなっている．また，ノード `dh`，`dm`，`ds` は `Set` 状態以外から参照できず，他の状態の定義に影響しないことが明示されている．

4.3　階層的な状態遷移の表現

図 10 に対して 3 章同様にストップウォッチ機能を追加する拡張を行う．ストップウォッチモードもカウンタの動作状態と停止状態とで振舞いが変化する．この状態遷移を管理するため，**図 11** のように switch モジュールを定義する．このモジュールはボタン入力 `button` と前回の更新からの経過時間 `dt` を入力にとり，ストップウォッチのカウント `stopWatch` とモードの脱出フラグ `exit` を定義している．

このモジュールを利用して図 10 に**図 12** のような `StopWatch` 状態を追加する．状態の定義中で `StopWatch` モ

```
switchmodule Watch
in button(Button(False,False,False)) : Button,
   curTime(Time(0,0,0)) : Time
out display(Time(0,0,0)) : Time,
    curTimeUpdate(NoneT) : OptTime
use Std, Watchlib
init Display

type SetPos = Hour | Min | Sec

state Display {
  ...
}

state Set(pos : SetPos) {
  node init[0] dh =
    pos of Hour -> 1, _ -> 0
  ...

  node display =
    if buttonB(button) then
      addTime(display@last, dh, dm, ds)
    else Retain

  node curTimeUpdate =
    pos of
      Sec -> if buttonA(button) then
               SomeT(display)
             else NoneT,
      _ -> NoneT

  switch:
    if buttonA(button) then
      pos of
        Hour -> Set(Min),
        Min -> Set(Sec),
        Sec -> Display
    else Retain
}
```

図 10　Switch 拡張を用いた時計システムの実装
Fig. 10　The basic digital watch using switch-extension.

```
switchmodule StopWatch
in button(Button(False,False,False)) : Button,
   dt(0) : Int
out stopWatch(Time(0,0,0)) : Time,
    exit(False) : Bool
use Std, Watchlib
init Stop

state Stop {
  ...
  node exit = buttonC(button)

  switch:
    if buttonB(button) then
      Run
    else Retain
}

state Run {
  ...
  node exit = False

  switch:
    if buttonB(button) then
      Stop
    else Retain
}
```

図 11　ストップウォッチモジュールの実装
Fig. 11　Implementation of a stopwatch module.

© 2020 Information Processing Society of Japan

```
state StopWatch {
  newnode display, exit =
    StopWatch(button, dt)

  node init[NoneT] curTimeUpdate = NoneT

  switch:
    if exit then Display else Retain
}
```

図 12　ストップウォッチモード状態の定義
Fig. 12　Definition of the stopwatch state.

ジュールをインスタンス化し，出力ノードや状態遷移の定義に利用している．状態定義内部で利用しているモジュールは，状態が切り替わった際に初期化される．この例では，StopWatch 状態に遷移したタイミングで StopWatch モジュールの内部状態が Stop になり，定義されているすべてのノードは指定した初期値で初期化される．

階層的な状態遷移は switch モジュールをサブモジュールとして利用することによって表現できる．ストップウォッチの状態に依存する振舞いを StopWatch モジュールにまとめ，curTimeUpdate のようにストップウォッチの実装と無関係な定義を上位のモジュールに記述している．このように状態遷移に階層を設けることによって状態間に共通する振舞いをまとめて記述できるようになる．さらに，状態をグループ化して扱うことによって見通し良くモジュールを記述することが可能になる．

4.4　状態の拡張性

ストップウォッチモードの実装の際，StopWatch 状態の追加以外に必要となる既存のコードに対する変更は次の 2 点のみである．1 点目は StopWatch モジュールの入力に必要となる dt を用意するための変更である．Watch モジュールの入力ノードに dt を追加し，上位のモジュールから dt をわたしてもらう必要がある．2 点目は Watch モジュールの Display 状態から StopWatch 状態への遷移を追加するための変更である．これについては Display 状態の定義の switch 節のみを修正すればよい．いずれの変更についても，既存の状態の定義を大きく修正することなく実現できているといえる．

このように，switch 拡張を用いることで状態ごとの振舞いの記述の独立性を高めることができる．ある状態の定義が他の状態の定義に影響を及ぼすことはないため，状態の追加によって既存の定義が変化してしまうことはない．状態遷移を拡張する際は新たに定義した状態への遷移を追加する変更のみを行えばよいため，既存のコードに対する影響を抑えて機能拡張を行うことができる．

5.　実装

前章で紹介した switch 拡張は純粋な Emfrp への変換を通じてサポートされている．本章では switch 拡張から純粋な Emfrp への変換について説明する．

```
switchmodule Module
in input(c_i) : T
out output(c_o) : T
use Std
init State_init

state State1 {
  # internal node
  node init[c_a] internal = expr_a

  # output node
  node output = expr_o

  switch:
    expr_s
}

state State2 {
  ...
}
```

図 13　変換前の switch モジュール
Fig. 13　A switch module (before translation).

```
module Module
in input(c_i) : T
out output : T
use Std

type State = State1 | State2

# for State1
node init[c_a] state1Internal =
  state@last of:
    State1 -> L(expr_a)
    _ -> state1Internal@last

node init[c_o] state1Output =
  state@last of:
    State1 -> L(expr_o)
    _ -> state1Output@last

node init[State_init] state1State =
  state@last of:
    State1 -> L(expr_s)
    _ -> state@last

# for State2
...
# output and state management
node init[c_o] output =
  state@last of:
    State1 -> state1Output
    State2 -> state2Output

node init[State_init] state =
  state@last of:
    State1 -> state1State
    State2 -> state2State

node init[True] switch =
  (state@last, state) of:
    (State1, State1) -> False
    (State2, State2) -> False
    _ -> True
```

図 14　純粋な Emfrp に変換された図 13
Fig. 14　Figure 13 translated in plain Emfrp.

5.1　方針

図 13 のように記述された switch モジュールは**図 14** のように変換される．定義された状態に対応して State とい

© 2020 Information Processing Society of Japan

うデータ型が定義され，状態の管理はこのデータ型を通して行われる．状態管理を行うためのノードとしてモジュールの状態を管理する state と状態の切り替わりを管理する switch が定義される．状態に対する条件分岐は更新開始時のモジュールの状態を指す state@last を用いて行う．一方，switch は内部ノードの管理に用いられる．

状態に対して定義されたノードや状態遷移はそれぞれ対応するノードへと展開される．各ノード定義はそれぞれ対応するノードへと変換される（*state1Internal*, *state1Output*）．これらのノードは mode@last が自身が定義されている状態と一致している場合のみ時変値の更新を行い，それ以外は前回の値を保持するという定義になっている．このような定義によって該当する状態以外では計算を行わないという振舞いを擬似的に表現している．ただし，この変換では必ずノードの直前値が参照されることになるため，switch 拡張ではすべてのノードに初期値を要求している．ノードの定義式は 5.4 節で説明するノード参照の変換（$\mathcal{L}(\cdot)$）が行われた後に展開先ノードに埋め込まれる．ノードが出力ノードだった場合，モジュールヘッダを参照して初期値を指定する変換も合わせて行う．さらに，switch 節についても遷移先状態を管理するノード（*state1State*）へと同様に変換される．初期値はヘッダの **init** を参照して指定される．

実際に出力となるノードと state ノードは各状態定義に対応して展開されたノードを用いて定義される．これらのノードは mode@last に応じて参照するノードを切り替えるように定義されている．また，switch ノードは state ノードの現在値と直前値を比較して状態の切り替わりを検知する定義になっている．

5.2　サブモジュールの処理

前節の変換はモジュールの依存関係のトポロジカル順でモジュールごとに行っている．依存するモジュールの変換を事前に行うことによって，2.3 節で説明した手法と同様にサブモジュールを処理することができる．モジュール内部に含まれるノードはその場に展開され，switch モジュールにおけるノード定義で **newnode** 記法を用いた場合にはモジュール内部のノードが状態定義の内部ノードとして扱われる．

5.3　直前値ノード

変換においては出力ノードと内部ノードで @last で参照する直前値の扱いが異なっている点に注意する必要がある．出力ノードはすべての状態定義間で共通するノードとして扱われる．初期化は最初のイテレーションが始まる前にのみ行われ，状態が切り替わった際の直前値は以前の状態で計算された値を引き継ぐことになる．一方，内部ノードは状態間で共有されることはなく，定義されている状態に切り替わるたびに初期化される．変換の際にはこれらの振舞いを表現する必要がある．

状態の切り替わりに合わせて内部ノードを初期化する処理は，直前値を管理するノードを別途定義することによって再現している．このノードを直前値ノードと呼ぶことにする．直前値ノードはノード switch を用いて定義されており，状態が切り替わった際に本体ノードの直前値をノードの初期値で上書きするような振舞いをする．たとえば，図 13 におけるノード *internal* の直前値ノードは以下のように定義される．

```
node init[c_l] state1Internal_last =
  if switch@last then c_l
  else state1Internal@last
```

ただし，すべての内部ノードに対して直前値ノードを定義してしまうとモジュールのネストの深さに対して指数的にノード数が増えてしまうため，@last による参照が存在する内部ノードのみ直前値ノードを用意している．

5.4　ノード参照の変換

Emfrp モジュールへの変換を行う際，出力ノードと内部ノードについては参照の変換を行う必要がある．この変換は図 **15** にまとめられる．**Retain** の変換もここで行われており，自分自身の直前値を指すように定義されている．

現在値に対する参照については各状態の定義で更新された後のものを参照する必要があるため，出力ノードも内部ノードも展開先のノードを参照するように変換している．

一方，直前値の参照については出力ノードと内部ノードで異なった変換を行っている．出力ノードは状態間で値が引き継がれるため，直前値の参照はそのまま出力ノードのものを参照するようにしている．内部ノードの直前値は状態の切替えによる初期化を反映する必要があるため，直前値ノードの参照へと変換している．

5.5　変換後コードにおけるノードの依存関係

すでに述べたように，Emfrp ではノードの現在値に対する依存関係に循環がないことを要求している．各状態の定義において循環する依存関係がなければ，変換後のコードにおいても循環する依存関係が存在しないことを簡単に説明する．変換後のコードについて，状態定義の展開の際に定義されたノードをローカルノードと呼び，それ以外のコードをグローバルノードと呼ぶことにする．

まず，ローカルノード内の依存関係を考える．異なる状態のローカルノードとの間に依存関係はなく，仮定よりノード定義に対応して展開されたノード間にも循環する依存関係は存在しない．また，switch 節を管理するノードは他のローカルノードから参照されることはなく，直前値ノードも定義内に @last による参照しか存在しないため，これらのノードは循環する依存関係には含まれない．以上からローカルノード間で循環する依存関係は存在しない．

© 2020 Information Processing Society of Japan

e	::=		（式）
	|	l	（リテラル）
	|	x	（定数）
	|	n	（ノード）
	|	n@last	（直前値）
	|	Retain	（自身の直前値）
	|	$(e, ..., e)$	（タプル）
	|	$f(e, ..., e)$	（関数適用）
	|	$C(e, ..., e)$	（バリアント）
	|	if e then e else e	（if 式）
	|	e of p -> $e, ..., p$ -> e	（パターンマッチ）
p	::=		（パターン）
	|	_	（ワイルドカード）
	|	l	（リテラル）
	|	x	（パターン変数）
	|	$(p, ..., p)$	（タプル）
	|	$C(p, ..., p)$	（バリアント）

$\mathcal{L}(l)$	=	l
$\mathcal{L}(x)$	=	x
$\mathcal{L}(n)$	=	n （入力ノードの場合）
$\mathcal{L}(n)$	=	n の展開先ノード （内部/出力ノードの場合）
$\mathcal{L}(n$@last)	=	n@last （入力/出力ノードの場合）
$\mathcal{L}(n$@last)	=	n の直前値ノード （内部ノードの場合）
$\mathcal{L}$(Retain)	=	$\mathcal{L}$(定義するノード@last) （ノード定義式の場合）
$\mathcal{L}$(Retain)	=	state@last （switch 節の式の場合）
$\mathcal{L}((e_1, ..., e_n))$	=	$(\mathcal{L}(e_1), ..., \mathcal{L}(e_n))$
$\mathcal{L}(f(e_1, ..., e_n))$	=	$f(\mathcal{L}(e_1), ..., \mathcal{L}(e_n))$
$\mathcal{L}(C(e_1, ..., e_n))$	=	$C(\mathcal{L}(e_1), ..., \mathcal{L}(e_n))$
$\mathcal{L}$(if e_1 then e_2 else e_3)	=	if $\mathcal{L}(e_1)$ then $\mathcal{L}(e_2)$ else $\mathcal{L}(e_3)$
$\mathcal{L}(e$ of p_1 -> $e_1, ..., p_n$ -> $e_n)$	=	$\mathcal{L}(e)$ of p_1 -> $\mathcal{L}(e_1), ..., p_n$ -> $\mathcal{L}(e_n)$

図 15　ノード参照の変換
Fig. 15　Translation rules for node references.

次に，グローバルノード内の依存関係を考える．グローバルノードは switch ノードを除いてローカルノードの現在値と state ノードの直前値のみを参照する定義になっている．また，switch ノードについては他のノードから現在値が参照されることはない．以上からグローバルノード間にも循環する依存関係は存在しない．

最後にグローバルノードとローカルノードの間の依存関係を考える．ローカルノードにおいては出力ノードに対する現在値の参照がすべてローカルノードのものに置き換えられており，state や switch に対する参照も直前値のものに限られる．よってローカルノードからグローバルノードの現在値が参照されることはないため，グローバルノードとローカルノードにまたがって存在するような循環する依存関係も存在しない．

以上から，各状態の定義において循環する依存関係がなければ，変換後のコードにおいても循環する依存関係が存在しないことが分かる．

表 1　Emfrp コードの比較
Table 1　Comparison of Emfrp code sizes.

	ノード数	行数
(a) 拡張前の Emfrp	19	98
(b) switch 拡張利用	26	135
(c) 変換で生成されたコード	51	304

表 2　C++コードの比較
Table 2　Comparison of genarated C++ code sizes.

	行数	TEXT	DATA	BSS
拡張前の Emfrp	700	2,172 B	392 B	448 B
switch 拡張利用	1,603	4,879 B	432 B	1,144 B
オーバヘッド	129%	125%	10%	155%

5.6　オーバヘッドに関する実験

拡張の利用や switch 拡張前の Emfrp への変換にともなうコードサイズやパフォーマンスについてのオーバヘッドを調べるためいくつかの実験を行った．本論文で扱った例題を対象に，3 章の実装と 4 章の実装の比較を行った．

まず，Emfrp コードのサイズについて比較を行った．(a) 3 章で示した拡張前の Emfrp を用いた実装，(b) 4 章で示した switch 拡張を利用した実装，(c) 4 章の実装を変換して生成した Emfrp コードの 3 つを比較した結果，**表 1** のようになった．(b) のコードサイズは (a) の 1.4 倍程度となった．これは (a) では 1 つのノードとして定義されていた振舞いが (b) では複数の状態に分かれたことが主な要因だと考えられる．一方，(c) のコードサイズは (b) の 2 倍前後となった．これには直前値ノードの定義が大きく影響していると考えられる．

Emfrp コードから生成された C++コードにおけるオーバヘッドについても調査した．3 章の実装と 4 章の実装からそれぞれ C++コードを生成し，コードの行数とコンパイルしたオブジェクトファイルにおける各セクションのサイズを比較をした．コンパイルは GCC version 5.5.0（x86）を用い，コードサイズに関する最適化を行う `-Os` オプションを付加して行った．結果は**表 2** のようになった．コード行数および TEXT セクションは 2.3 倍程度増加し，Emfrp コードのサイズにおける増加量よりやや少なめになった．一方，BSS セクションは 2.6 倍程度増加し，Emfrp コードのサイズと同程度の増加量となった．

さらに，ノードの更新時間に関するオーバヘッドを調べる実験も行った．3 章の実装と 4 章の実装から生成された C++コードを編集し，更新を 10^8 回繰り返した後停止す

© 2020 Information Processing Society of Japan

表 3　10^8 回の更新にかかった時間
Table 3　Elapsed times for 10^8 updates.

拡張前の Emfrp	5.43 s
switch 拡張利用	13.80 s

るようにそれぞれ変更した．これらに対して同様の入出力コードを追加してコンパイルし，Intel Core i5（2.4 GHz）上で実行した．なお，コンパイルは先程の実験と同様に行った．実行時間を `time` コマンドで計測し，10 回の実行のユーザ CPU 時間を平均したところ**表 3** のようになった．実行時間の増加量も 2.5 倍程度となり，ノード数の増加量と同程度になった．これは今回の例題ではそれぞれのノードの更新にかかる時間にほとんど差がなく，各イテレーションでは定義されたすべてのノードが更新されるためだと考えられる．

6.　関連研究

本研究が提案した switch 拡張は Yampa [7] の `switch` オペレータに強く影響を受けている．Yampa は Arrowised FRP をサポートする Haskell のライブラリである．Arrowised FRP では時変値ではなく時変値上の関数であるシグナル関数を第 1 級オブジェクトとして扱う．Yampa では switch コンビネータと呼ばれるオペレータ群を通して適用するシグナル関数を動的に切り替えることができる．このうち最も単純なオペレータが以下のような型を持つ `switch` オペレータである．

```
switch :: SF a (b, Event c)
  -> (c -> SF a b)
  -> SF a b
```

ここで `SF a b` は `a` 型の時変値から `b` 型の時変値へのシグナル関数の型であり，`Event c` は `c` の型のイベントを表現する型である．第 1 引数には切替え前の振舞いを表現するシグナル関数を渡す．切替え前はこのシグナル関数に従ってタプルの第 1 要素を返すようなシグナル関数として振る舞う．シグナル関数の切替えはタプルの第 2 要素がイベントを送出した際に起こる．第 2 引数の関数を用いて切替え先のシグナル関数を計算し，以降はそのシグナル関数として振る舞う．

状態遷移を含むコンポーネントも `switch` コンビネータによって表現できる．たとえば，例題のストップウォッチモードは**図 16** のように表現できる．Yampa に組み込まれている型との衝突を防ぐため，例題における `Time` 型が `WTime` 型へとリネームされている点に注意されたい．`sfSfop` と `sfRun` は，それぞれ停止状態と動作状態における振舞いを表現するシグナル関数を返す．状態間でストップウォッチのカウントを引き継ぐため，引数にカウントの初期値を受け取りシグナル関数に埋め込めるようになっている．状態遷移は出力のタプルの第 2 要素で管理しており，

```
-- in  button : Button, dt : Int
-- out stopWatch : WTime, exit : Bool

sfStop :: WTime
  -> SF (Button, Int) ((WTime, Bool), Event WTime)
sfStop cnt =
  …

sfRun :: WTime
  -> SF (Button, Int) ((WTime, Bool), Event WTime)
sfRun cnt =
  …

sfStopWatch :: SF (Button, Int) (WTime, Bool)
sfStopWatch = stateStop (WTime 0 0 0)
  where
    stateStop cnt = dSwitch (sfStop cnt) stateRun
    stateRun cnt = dSwitch (sfRun cnt) statenStop
```

図 16　Yampa によるストップウォッチモードの実装
Fig. 16　Implementation of the stopwatch mode in Yampa.

直前のカウントをイベントとして送出することで遷移を指示する．これらの関数を利用してストップウォッチ機能全体を表現するシグナル関数 `sfStopWatch` を定義している．ここでは `dSwitch` オペレータを用いて計算の切替えを行っている．このオペレータは `switch` と同様のオペレータである．ただし，シグナル関数の切替えが起こった瞬間の出力の扱いが異なっており，`switch` では切替え後のシグナル関数の出力を返すが，`dSwitch` では切替え前のシグナル関数の出力を返す．多段階の切替えは，切替え先のシグナル関数として `switch` や `dSwitch` で定義されたシグナル関数を指定することで実現できる．例では `stateStop` と `stateRun` の相互再帰で永続的な状態遷移を表現している．

しかし，Haskell のライブラリである Yampa をそのまま小規模組込みシステムの開発へ適用するのは困難である．本研究の拡張では，切り替える先の振舞いを静的に定義された状態集合に制限することで動的なメモリ管理を行うことなく実行時の計算の切替えを実現している．Yampa にはシグナル関数のイベントを送出してシグナル関数の切替えを行う `rSwitch` オペレータがある．一方，Emfrp では Yampa のシグナル関数に相当する第 1 級オブジェクトは提供されておらず，`rSwitch` のように切替え先のシグナル関数が限定されないオペレータは表現できない．今回導入した switch モジュールは，状態ごとのノード定義によってシグナル関数を表現し，switch 節で切替えイベントを管理することで `dSwitch` 相当の機構を提供している．

Emfrp において実行時の計算の変化を可能にする拡張としては，文脈指向プログラミング（COP）による拡張 [16] が提案されている．文脈（実行時情報）に依存する振舞いを層と呼ばれる機構にまとめ，実行時に適切な層を活性化させることによって既存のノード定義を上書きすることができる．ただし，COP 拡張では層にローカルなノードを定義する機構は提供されておらず，状態に固有な振舞いを記述することはできない．これらの性質からステートマシンに基づく計算の切替えを抽象化する機構としては適してい

© 2020 Information Processing Society of Japan

ない．一方で，状態間で共通する振舞いを多く持ち，一定の条件でノードの定義を一時的に変更したい場合においては COP 拡張の方が適している．このようなケースの記述を switch 拡張で支援することは今後の課題となっている．

計算資源の限られた環境におけるリアクティブシステムの開発を支援する言語・ライブラリは文献 [6], [8], [9], [13] で提案されているものなどがある．これらの多くは状態遷移を明示的にあつかう言語機構を持たないが，Emfrp と同じく小規模組込みシステムを対象とした FRP 言語である Juniper [6] では高階の時変値を扱うことで時変値の依存関係を動的に変化させることができる．高階の時変値を扱う際には現在参照されていない時変値の処理が問題となり，空間漏れ・時間漏れの原因となることが多いが，Juniper では時変値が参照されなくなったタイミングで値の保存と更新を止めることによってこれらを解決している．しかし，動的なメモリ管理を避けるという Emfrp の設計にのっとったまま高階の時変値をサポートすることは困難である．

Statechart [5] はシステムの状態遷移を設計するツールである．Statechart では階層的な状態や直交する状態を表現できる．4.3 節で述べたように，階層的な状態は switch モジュールをネストさせて用いることによって表現できる．また, 直交する状態についても複数の switch モジュールを同時に利用することによって表現できる．同じく Statechart に影響を受けた言語にオブジェクト指向言語 Plaid [15] がある．Plaid はオブジェクトの状態ごとに利用できるメソッドを定義し，メソッド定義内で状態遷移の管理を行う．これによって，状態に応じてアクセス可能な API が切り替わるオブジェクトを表現できる．Plaid では階層的な状態や直交する状態の表現に加え，直交する状態の連動した遷移を表現することが可能になっている．一方，switch 拡張では状態の遷移はモジュール内に閉じた管理をされており，他のモジュールの状態遷移と連動した振舞いを表現するためにはモジュールの入出力を通して現在の状態を受け渡す必要がある．このような設計は状態遷移の記述力を落としているが，モジュールの独立性を高く保つことによってテストやデバッグを容易にしている．

組込みシステム開発では MATLAB/Simulink [10] などの CAE/CAD ツールを用いたモデルベース開発（MBD）が主流となっている．Stateflow [11] は MATLAB/Simulink で利用可能なグラフィカルなプログラミング言語であり，Statechart と同様の状態遷移の表現やそれにともなう制御ロジックの切替えもサポートしている．Stateflow のセマンティクスは直交する状態や上位階層の状態から送信されるイベントなどから影響を受けるが，Emfrp は純粋関数型言語であり，導入した switch モジュールも振舞いは参照するノードにのみ依存する．そのため，Stateflow と比較してコンポーネントの独立性はより高くなっているといえる．

既存の FRP 言語・システムでプログラムの状態遷移を明示的に書くことのできる機構としては，上にあげた Yampa の switch コンビネータ，高階時変値，COP 拡張がある．以下，これらによる状態遷移の記述と本研究で提案する switch モジュールによる記述との比較を行う．まず switch コンビネータと高階時変値であるが，これらは切り替える先の振舞いを自由に指定できるため，それが状態遷移を意図して書かれたものかどうかはプログラムの文面から判断することは難しい．COP 拡張も同様の問題を持つことに加えて，活性化する層の組合せによって振舞いを変化させることができるため，状態数が爆発するような記述も可能になる．一方 switch モジュールによる記述は状態遷移を専用の構文で記述する．この記述では状態数の爆発も生じず，また検証やテストを目的としてプログラムから状態遷移動作を自動的に抽出することも容易である．

7. まとめ

小規模組込みシステム向け FRP 言語 Emfrp に対して状態遷移による振舞いの変化の記述に特化した言語機構を導入する拡張を行い，状態依存動作を含むコンポーネントのモジュール性を向上させた．本研究で提案した switch 拡張は，状態ごとに時変値と状態遷移の定義を行う機構を提供することで，FRP によるリアクティブシステムの宣言的な記述とステートマシンによる状態遷移の直感的な管理を両立させている．また，各状態の定義の独立性を高めることで可読性・拡張性の高いコードを記述することを可能にしている．さらに，switch 拡張は（拡張を導入する前の）Emfrp への変換として実装されているため，プログラムが実行時に使用するメモリ量が静的に決定できることなどの，小規模組込みシステムへの利用に有利な性質は保たれている．

今後の展望としてはまず処理系の改善があげられる．5 章で説明した実装は，状態定義を展開したノードすべてに更新がかかるためオーバヘッドが大きくなってしまうという問題点を持つ．Emfrp の実行モデルを拡張し，該当する状態のノードのみを更新するようなコードを生成することができればパフォーマンスを向上できると考えている．また，表現力についてもさらなる拡充を予定している．6 章であげた COP 拡張が得意とするケースのほかに，処理の一時的な中断など状態が切り替わっても値を保存したいケースについても対応が必要である．これに対しては，出力ノードの他にも状態間で共有できるノードを用意することを考えている．加えて，switch モジュールから抽出した状態遷移動作から時間オートマトン [1] などの形式的なモデルを生成し，モデル検査などによる検証を支援することも検討している．

謝辞　改稿にあたり査読者および東京工業大学の森口草介氏より有益な助言およびコメントをいただいた．ここに感謝する．本研究の一部は JSPS 科研費 18K11236 の助成を受けている．

© 2020 Information Processing Society of Japan

参考文献

[1] Alur, R. and Dill, D.L.: A Theory of Timed Automata, *Theoretical Computer Science*, Vol.126, No.2, pp.183–235 (1994).

[2] 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, pp.52:1–52:34 (2013).

[3] Czaplicki, E. and Chong, S.: Asynchronous Functional Reactive Programming for GUIs, *34th ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp.411–422 (2013).

[4] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN Intl. Conf. Functional Programming*, pp.263–273 (1997).

[5] Harel, D.: Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, Vol.8, No.3, pp.231–274 (1987).

[6] Helbling, C. and Guyer, S.Z.: Juniper: A Functional Reactive Programming Language for the Arduino, *4th Intl. Workshop on Functional Art, Music, Modelling, and Design*, pp.8–16, ACM (2016).

[7] Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming, *Advanced Functional Programming*, LNCS, Vol.2638, pp.159–187, Springer (2003).

[8] Kaiabachev, R., Taha, W. and Zhu, A.: E-FRP with Priorities, *7th ACM/IEEE Intl. Conf. Embedded Software*, pp.221–230 (2007).

[9] Mainland, G., Morrisett, G. and Welsh, M.: Flask: Staged Functional Programming for Sensor Networks, *13th ACM SIGPLAN Intl. Conf. Functional Programming*, pp.335–346 (2008).

[10] MathWorks: Simulink – シミュレーションおよびモデルベースデザイン – MATLAB & Simulink，入手先 ⟨https://jp.mathworks.com/products/simulink.html⟩.

[11] MathWorks: Stateflow – MATLAB & Simulink, available from ⟨https://jp.mathworks.com/products/stateflow.html⟩.

[12] Pembeci, I., Nilsson, H. and Hager, G.: Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages, *4th Intl. ACM SIGPLAN Conf. Principles and Practice of Declarative Programming*, pp.168–179 (2002).

[13] Sant'anna, F., Ierusalimschy, R., Rodriguez, N., Rossetto, S. and Branco, A.: The Design and Implementation of the Synchronous Language CÉU, *ACM Trans. Embedded Computing Systems*, Vol.16, No.4, pp.98:1–98:26 (2017).

[14] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *Companion Proc. 15th Intl. Conf. Modularity*, pp.36–44, ACM (2016).

[15] Sunshine, J., Naden, K., Stork, S., Aldrich, J. and Tanter, É.: First-Class State Change in Plaid, *26th Annual ACM SIGPLAN Conf. Object-Oriented Programming Systems Languages and Applications*, pp.713–732 (2011).

[16] Watanabe, T.: A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems, *10th Intl. Workshop on Context-Oriented Programming*, pp.23–30, ACM (2018).

松村 有倫

2018 年東京工業大学情報理工学院情報工学系卒業．現在，同大学情報理工学院情報工学系情報工学コース修士課程在学．プログラミングパラダイムや計算機言語の抽象化機構に興味があり，現在はリアクティブプログラミングにおけるコードのモジュール化についての研究に従事．

渡部 卓雄　（正会員）

1986 年東京工業大学理学部情報科学科卒業．1991 年同大学大学院理工学研究科情報科学専攻博士課程修了．理学博士．日本学術振興会特別研究員，イリノイ大学計算機科学科研究員，北陸先端科学技術大学院大学情報科学研究科助教授，東京工業大学大学院情報理工学研究科計算工学専攻准教授をへて，現在，東京工業大学情報理工学院情報工学系教授．メタプログラミングと自己反映計算，形式手法，プログラミング言語の研究に従事．

© 2020 Information Processing Society of Japan