小規模組込みシステム向けFRP言語における非同期タスク処理機構

横山 陽彦[1]，森口 草介[1]，渡部 卓雄[1]

[1] 東京工業大学 情報理工学院 情報工学系
akihiko@psg.c.titech.ac.jp, {chiguri,takuo}@acm.org

概 要　小規模組込みシステム向け関数リアクティブプログラミング (FRP) 言語 Emfrp は適切な機能制限を行うことで，時変値更新処理の停止性判定や実行時に必要なメモリ量の静的な決定が可能である．そのため，リソースの制限された環境においても安全にリアクティブな動作を行うことが可能である．一方でその機能制限のため，例えば外部から得られた情報をもとにグラフ構造の構築や探索を行うなど，比較的時間のかかる処理 (heavy-task) の記述は想定されていない．また，Emfrp は同期的に処理を実行するため，heavy-task を外部関数呼び出しとして愚直に記述すると入力に対する応答性が悪化してしまう．一般のプログラミング言語では，このような処理を非同期処理として分離することで記述性と応答性を高めている．そこで本研究では，並行プログラミングで一般に用いられている future や promise に類似した機構を Emfrp に導入することで，従来の時変値更新によるリアクティブな動作を保ちながら，非同期的 heavy-task が実行可能な実行基盤の構築を目指す．本稿では，Emfrp に対する heavy-task 実行機構のナイーヴな導入に対する問題点を挙げたのち，例題ベースで提案手法を述べ，言語ランタイム実装についての検討を行う．

1　はじめに

リアクティブシステムとは，外部からの非同期的な入力に反応し内部状態を変化させながら応答し続けるシステムを指す．一般にリアクティブシステムのプログラミングにはポーリングやコールバック，割り込み処理などの技法がよく使われている [2]．しかしながら，それらの技法によりプログラム全体の見通しが悪くなりやすい．**関数リアクティブプログラミング (Functional Reactive Programming, FRP)** は，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．具体的には，時間とともに変化する値を抽象化した**時変値 (time-varying value)** を組み合わせることで見通しよくリアクティブシステムを構築することを目指す．時変値同士のデータフローに注目しながらシステムを記述できることが FRP の特徴である．時変値による FRP はその登場以来，対話的アニメーション [5]，GUI[3]，Web アプリケーション [4]，ロボティクス [7, 11]，組込みシステム [16, 6]，IoT[15] 等の様々な分野で研究され，その有用性が示されてきた．

Emfrp[16] は特に小規模組込み環境での実行を対象とした FRP 言語である．マイクロコントローラ等の CPU 及びメモリリソースに限りがある環境での使用を想定し，実行コードのメモリフットプリントが小さくなるように設計されている．同時に，デバッグの難しい組込み環境において，安全なリアクティブシステムの動作を目指し，時変値についての制約や構文や型システムに制限をかけることで，時変値更新処理の停止性や実行時に必要となるメモリ量の静的な決定を保証している．我々は Emfrp をベースとした複数の拡張 [19, 13, 12, 20] を通し，様々なアプリケーションに向けて研究を行ってきた．

小規模な組込みシステムであっても，規模の大きくないグラフ探索やラフな構文解析など，**ある程度複雑なデータ構造を対象とした比較的時間のかかる操作 (heavy-task と呼ぶ)** が要求されることは多い．そしてそのような操作がリアクティブシステムを構成する処理の一部として出現するこ

とは珍しくない．Emfrp は前述した機能制限により複雑なデータ構造を時変値として扱うことができない．また，その実行モデルのため，外部関数呼び出しとして heavy-task を行うとシステムの応答性が悪化し，リアクティブな振る舞いを保つことができなくなってしまう．

本研究では，前述の問題を解決し，**Emfrp のリアクティブな振る舞いと heavy-task 実行の協調を可能にする言語拡張及び言語ランタイム**の提案を行う．言語拡張に関しては，Emfrp 上で heavy-task を扱う言語機構に加え，並行プログラミングにおいて広く使用されている future や promise と類似した型を導入した．言語ランタイムに関しては，小規模組込み環境を意識してシングルコアマイコンで動作するような設計とした．

本論文の構成を次に示す．第 2 節にて，背景知識として拡張を行う前の Emfrp の概要を説明する．第 3 節にて，本研究の動機となる例を提示し，ナイーヴな解決策での問題点を示す．第 4.2 節では，本研究の提案手法として Emfrp に対する言語拡張と言語ランタイム拡張について述べ，それらの拡張を利用した動機となる例のプログラム記述を提示する．第 5 節にて既存研究との関連について論じたのち，第 6 節にてまとめと今後の課題について述べる．

2　小規模組込みシステム向け FRP 言語 Emfrp

Emfrp [16] 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．本節では，本研究での言語拡張を行う前の Emfrp について解説する．言語についてのより詳細な説明は文献 [16] やソースコードリポジトリ[1]を参照されたい．

2.1　例題

図 1 は，文献 [7] において Haskell の FRP ライブラリである Yampa の例題として挙げられているものを Emfrp で書き直したプログラムである．このプログラムは差動二輪ロボットの位置を計算しつづける．時刻 t におけるロボットの左右の車輪の対地速度をそれぞれ $v_l(t)$ と $v_r(t)$ とすると，その時刻におけるロボットの位置 $(x(t), y(t))$ と向き $\theta(t)$ は以下のような関係式になる．ただし，左右の車輪間の距離を l とする．

$$x(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u)) \cos\theta(u) du \tag{1}$$

$$y(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u)) \sin\theta(u) du \tag{2}$$

$$\theta(t) = \frac{1}{l}\int_0^t (v_r(u) - v_l(u)) du \tag{3}$$

Emfrp プログラムはモジュールという単位で記述される．図 1 では，ファイル `RobotPos.mfrp`とファイル `CalcPosY.mfrp`にてそれぞれ 1 つずつモジュールが定義されている．モジュールはヘッダ部と本体から構成される．ヘッダ部ではモジュール名，入出力ノードの定義，ライブラリのインポートが行われる (`RobotPos.mfrp` 2–8 行目，`CalcPosY.mfrp` 2–5 行目)．本体では中間ノード，出力ノードの定義，サブモジュールの展開，定数や関数の定義が行われる (`RobotPos.mfrp` 10–20 行目，`CalcPosY.mfrp` 7 行目)．

2.2　時変値 (ノード) と直前値

Emfrp では FRP における時変値はノードと呼ばれるオブジェクトで表される．`RobotPos`モジュールにおいて，`vl`，`vr`，`t`，`x`，`y`，`dt`，`theta`はノードである．これらはそれぞれ関係式中の $v_l(t)$，$v_r(t)$，t，$x(t)$，$y(t)$，Δt，$\theta(t)$ と対応する．ノードは入力ノード，出力ノード，中間ノードに分類される．`vl`，`vr`，`t`が入力ノード，`x`，`y`が出力ノード，`dt`，`theta`が中間ノードである．

[1] `https://github.com/psg-titech/emfrp`

```
# RobotPos.mfrp
module RobotPos    # モジュール名
in  vl : Float,    # 左車輪の対地速度 [m/sec]
    vr : Float,    # 右車輪の対地速度 [m/sec]
    t(0) : Int     # 経過時間 [msec]
out x : Float,     # ロボットのx 座標 [m]
    y : Float      # ロボットのy 座標 [m]
use Std, Params    # インポートするライブラリ名

# 直前値との差によって微小時間を計算 [sec]
node dt = (t - t@last) / 1000.0

# ロボットの傾き (角度) [rad]
node init [0.0] theta = theta@last + (vr - vl) * dt / l

# ロボットのx 座標 [m]
node init[0.0] x = x@last + (vr + vl) * cos(theta) * dt / 2.0

# ロボットのy 座標 [m]
newnode y = CalcPosY(vl, vr, theta, dt) # サブモジュール利用
```

```
# Params.mfrp
material Params

# 定数定義
data l = 0.1 # 車輪間距離 [m]

# 関数定義
func max(a: Float, b: Float) = if a > b then a else b
```

```
# CalcPosY.mfrp
module CalcPosY
in  vl : Float, vr : Float, theta: Float, dt : Float
out y : Float
use Std

node init[0.0] y = y@last + (vr + vl) * sin(theta) * dt / 2.0
```

図 1. Emfrp による差動二輪ロボットの位置計算プログラム

入力ノードは外部からの値が入力される．特にセンサーのような外部機器に接続されていることが想定されている．例えば RobotPosモジュールにおいては，vl，vrは車輪の回転数を計測するセンサーであるロータリーエンコーダーからの値が入力される．tにはシステムのタイマが接続され，システムの経過時間が入力される．

中間ノードと出力ノードは **node** $n = e$あるいは **node init**[c] $n = e$の構文によって更新式が定義される．n はノードの名前，e は式，c は定数である．RobotPosモジュールでは，11，14 行目にて中間ノード dt，thetaの定義を，17 行目にて出力ノード xの定義を行っている．

Emfrp の特徴として，n@last式によってノードの直前値を取得することができる．RobotPosモジュールの dtノードは@lastを使用することで，前回の時変値更新時刻と今回の時変値更新時刻の差から微小時間を算出している．直前値を利用することで，時変値の変化量や累積値を計算することが容易である．例えば，thetaノードでは，@lastを使用した簡潔な記述で時間積分を微小量の累積値として近似している．

直前値が参照されうるノードについてはその初期値が設定される．中間ノードと出力ノードに関しては **init**つきのノード定義初期値を設定する．入力ノードに関しては RobotPos.mfrp 5 行目のようにモジュールのヘッダ部に記述する．

ノード x の定義式中にノード y への参照が含まれる場合 (y@lastはノード y への参照とはみなさない)，ノード x はノード y に依存するという．Emfrp では時変値更新の順序を適切に定めるため

```c
// RobotPosMain.c
void Input(float* vl, float* vr, int* t){
    /* Fill Your Code */
    /* センサーから値を取得 */
}

void Output(float x, float y){
    /* Fill Your Code */
    /* 得られた値を出力 */
}

int main(void){
    ActivateRobotPos();
    return 0;
}
```

図 2. RobotPosモジュールのためのテンプレートコード (抜粋)

に，プログラム中の全てのノードについて，それらの依存関係が非巡回有向グラフ (DAG) であることが静的に検査される．

2.3 ライブラリとサブモジュール

Emfrp モジュールでは，RobotPos.mfrp 8 行目のように **use**の後に名前を指定することでライブラリ (マテリアルファイル) をインポートすることができる．RobotPosモジュールでは標準ライブラリ Stdの他にファイル Params.mfrpをライブラリとしてインポートしている．ライブラリファイルはマテリアルと呼ばれ，ファイル先頭の **material**によって識別される．マテリアルには，定数定義，関数定義，型定義を含めることができる．ここではロボットの車輪間距離 lを定数として定義し，RobotPosモジュールのノード thetaの定義式中で使用している．

定義されたモジュールはサブモジュールとして別のモジュール内部で利用することができる．予約語 **newnode**によって，サブモジュールの利用を区別する．例えば RobotPos.mfrp 20 行目では，CalcPosYモジュールがサブモジュールとして利用されている．RobotPosモジュールのノード vl，vr,theta，dtがサブモジュール CalcPosYへの入力となり，出力ノードが yに束縛される．サブモジュールは，コンパイル時に処理系内部で展開される．ネストしたモジュールは再帰的な展開が行われる．それぞれのモジュールに関しても，ノードの依存関係の制約と同じように循環した依存関係は禁止されている．

2.4 外部との入出力

Emfrp プログラムはコンパイラによって C 言語のソースコードへとトランスパイルされる．その際同時に，外部への入出力を行う関数とコンパイルされた Emfrp プログラムを起動するための関数呼び出しが記述されたテンプレートコードが出力される．図 2 は，RobotPosモジュールをトップレベルモジュールとしてコンパイルした際に出力されるテンプレートコードの抜粋である．Emfrp コンパイラはトップレベルモジュール RobotPosに対応した C 言語の ActivateRobotPos関数を生成する．main関数にて ActivateRobotPos関数を呼び出すことで，Emfrp プログラムが実行される．ActivateRobotPos関数の処理中に外部入力が必要になったときに Input関数が呼び出される．同様に，外部出力の際には Output関数が呼び出される．

2.5 実行モデル

Emfrp は push 型 [2] の時変値更新処理を行う．これは入力ノードの変化を依存するノードに順に伝搬させていく実行モデルである．ノードの依存関係にループが存在しないことがコンパイル時に検査されるため，依存関係の順で時変値更新処理を行うことで，適切に時変値更新処理を行うことができる．Emfrp では，入力関数の呼び出し，入力ノードの更新，依存関係の順に内部ノードと出

力ノードを更新，出力関数の呼び出し，直前値更新処理，メモリ管理の順に処理が行われる．この一連の処理をイテレーションと呼ぶ．このイテレーション処理を休みなく逐次的に繰り返すことで，リアクティブな振る舞いを行うことができる．

3 動機

本節では，本研究の動機となるリアクティブ動作と heavy-task 実行 (後述) の協調が必要なシステムを例に挙げ，Emfrp における言語拡張の必要性を説明する．

3.1 Emfrp の機能制限

Emfrp では，FRP 特有の問題である時間漏れ (time-leak) や空間漏れ (space-leak) の発生を防ぐため，及び「記述されたプログラムがメモリ不足による実行時エラーを起こさない」こと,「リアクティブな動作を続ける」ことを保証するために幾つかの機能的な制限を課している．例えば，時変値の過去値参照を直前値に制限する，時変値そのものを時変値として扱う高階時変値の禁止，関数やデータ型の再帰的定義の禁止などである．配列に関しても範囲外アクセスによるエンバグを嫌い導入されていない．この制限から，ノード更新式として記述できる式は比較的単純なものに限られる．そのため，それぞれのノード更新は一瞬のうちに終わるとみなすことができるので，イテレーションの連続実行により十分な応答性を担保することが可能である．

一方，これらの制限は同時に，グラフのようなある程度複雑なデータ構造の定義やそれを用いる操作の実現を困難にしている．これらのデータ構造は容易に使用メモリの肥大化を起こしうる他，ノード更新の計算が極端に遅くなることで，システム全体の応答性を落としてしまうためである．Emfrp に対しサイズ制限つき再帰データ型を導入した EmfrpBCT [20] では，最大サイズが固定されたリストや木構造の構築や操作が可能であるが，データ構造に対するインプレースな更新が行えないため，更新処理では複数のデータ構造が複製されてしまう．そのため，メモリリソースが極端に制限された環境では，利用可能なサイズを小さくせざるを得ないといった問題も発生する．

3.2 Heavy-task

小規模な組込みシステムにおいても，グラフのようなデータ構造を用いた処理が要求されることは多い．その中でも特に，イテレーション毎の実行が必須ではないが，最悪時間が比較的長く，同期して実行した場合に他の入力・出力の応答性を悪化させるようなタスクについて考える．本稿では，そのような操作を **heavy-task** と呼ぶ．

前節で説明した制限からもわかるとおり，heavy-task は応答性に関する問題としているものの，Emfrp(BCT) では記述が困難なものでもある．そのため，heavy-task の記述には FFI，Emfrp の場合は C への FFI を用いて記述する．ただし，ナイーヴに実装を許すと応答性の悪化が発生するため，十分な応答性をもつリアクティブな動作と処理時間のかかる heavy-task 実行とを協調するために，非同期的なタスク実行基盤が必要となる．

3.3 例題　迷路探索ロボット

リアクティブ動作と heavy-task 実行との協調が必要なシステムの例として，迷路を探索するロボットを取り上げる．この例はロボット競技「マイクロマウス」[2]の迷路探索を模したものである．センサーと駆動輪を持つロボットが，与えられた未知の迷路を探索し，スタート区画からゴール区画まで自律的に移動することを目的としている．

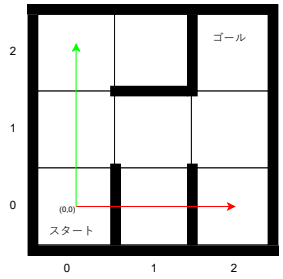

図 3. 迷路の一例

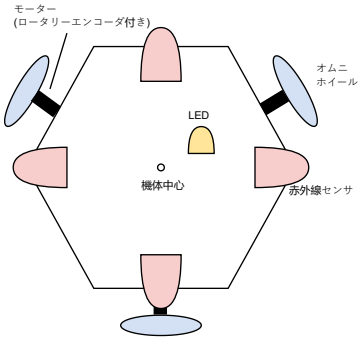

図 4. 全方向移動ロボット

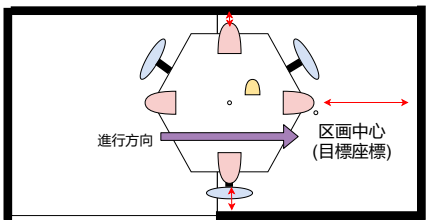

図 5. 目標区画への進入

3.3.1 問題設定

迷路全体は長方形であり，複数個の壁によって正方形の区画に分割されている．迷路の一例を図 3 に示す．迷路の外周は全て壁に囲まれている．ロボットには，迷路全体の大きさ，全区画の個数，区画の大きさ，スタート区画の位置，ゴール区画の位置，スタート区画の壁の配置はあらかじめ与えられている．スタート位置は迷路の左下で固定，スタート区画の壁は東南西の方向にあり北側に壁はない．ロボットにとって未知なのは迷路中の壁の情報である．

ロボットを上部から見た概要図を図 4 に示す．ロボットには壁の有無を判断するための赤外線センサーが 4 つ，オムニホイール[3]と呼ばれる特殊な車輪のついたモーター 3 つ，ゴールへ到着した旨を伝える LED1 つが接続されている．それぞれのモーターには回転数を計測できるロータリーエンコーダーが搭載されている．オムニホイールは通常の車輪の円周上に複数の車輪がついた形状をしており，車輪の回転方向と垂直な方向にも移動させることができる．このオムニホイールを図 4 のように配置し，それぞれモーター出力を適切に行うことで，ロボット本体を前後左右のどの方向にも移動させることができる．すなわち，今回の例で扱うロボットは全方向移動車であり，区画と区画の間を移動する際に機体の回転動作は行わないものとする．

このロボットがスタート区画からゴール区画にたどり着くための動作を説明する．まずロボットはスタート区画に置かれた状態でプログラムが開始される．スタート区画の壁配置は確定しているので，1 つ北側の区画を目標区画と定め，そこへ移動する．機体中心が目標区画の中心に移動したとき，機体が目標区画に到着したと判断し，目標区画の壁情報を記録する．今までに記録された壁情報をもとに，迷路探索アルゴリズムを実行し，次に移動すべき区画を得る．ただし次に移動すべき区画は現在の区画の東西南北のいずれかの方向に 1 区画だけずれた区画である．迷路探索アルゴリズムには A*アルゴリズムや拡張左手法などが使用できる．アルゴリズムによって得られた区画を新しい目標地点として定め，そこへ移動する．この一連の動作を複数回繰り返すことでロボットは最終的にゴール区画へたどり着く．ゴール区画へ辿り着いた時ロボットは LED を点灯させ，実行が終わったことをユーザーに通知する．

この一連の動作は，機体を目標区画へ移動させる動作，未知の壁情報を内部に記録する動作，内部に記録された壁情報と迷路探索アルゴリズムによって次に向かうべき区画を計算する動作の 3 つに分解することができる．ロボットを目標区画へ移動させるという動作はモーターの回転数，目標座標，経過時間を入力，現在の機体位置を内部状態，モーターへの出力値を出力としたリアクティブな動作として見なすことができる．一方で，壁情報を記録する動作と次に向かうべき区画を計算する動作は，迷路を表すグラフ構造に変更を加え，そのグラフを使用した探索アルゴリズムを行うため，それぞれ heavy-task と見なすことができる．

[2] https://www.ntf.or.jp/?page_id=25
[3] https://en.wikipedia.org/wiki/Omni_wheel

```
#  PID 制御による全方向移動車の位置制御
module MovePID
in tar_x : Float, tar_y : Float, # 目標座標
   enc1: Int, enc2: Int, enc3: Int, # 車輪の前回からの相対回転数
   duration : Float # イテレーションにかかった時間 (秒)
out distance : Float, # 目標地点までの直線距離
    duty1 : Int, duty2 : Int, duty3 : Int # 車輪に出力するduty 比
use Std

data Kp = ... # PID 制御における比例項の係数
data Ki = ... # PID 制御における積分項の係数
data Kd = ... # PID 制御における微分項の係数

# それぞれのモーターの回転数から中心座標の移動量を計算する
func motor_to_vector(...) = ...

# 中心座標の移動量からそれぞれのモーターへの出力を計算する
func motor_speed(...) = ...

# 現在位置の更新
node (dx, dy) = motor_to_vector(enc1, enc2, enc3)
node init [(0, 0)] (cur_x, cur_y) = (cur_x@last + dx, cur_y@last + dy)

# X 座標についての目標値との誤差
node init [0] e_x = tar_x - cur_x
# X 座標についての目標値との誤差の積分 (台形積分)
node init [0] ei_x = ei_x@last + (e_x + e_x@last) / 2 * duration
# X 座標についての目標値との誤差の微分
node ed_x = if duration == 0 then 0 else (e_x - e_x@last) / duration
# X 座標についての PID 制御による制御量
node delta_x = Kp * e_x + Ki * ei_x + Kd * ed_x

# Y 座標についての計算(X 座標と同様)
node init [0] e_y = tar_y - cur_y
node init [0] ei_y = ei_y@last + (e_y + e_y@last) / 2 * duration
node ed_y = if duration == 0 then 0 else (e_y - e_y@last) / duration
node delta_y = Kp * e_y + Ki * ei_y + Kd * ed_y

# 目標地点までの直線距離
node distance = sqrt(e_x*e_x + e_y*e_y)
# それぞれのモーターへの出力
node (duty1, duty2, duty3) = motor_speed(delta_x, delta_y)
```

図 6. PID 制御による全方向移動車の位置制御モジュール

前述したロボットの動作では目標区画への移動，壁情報の記録，次の目標区画の設定がそれぞれ順に行われていた．しかし，図 5 に示すように，ロボットの機体中心が目標区画の中心にある程度近い場合には，その目標区画の壁情報を得ることができる．そのため，ロボットが目標地点へと向かう動作と並行して壁情報を記録し，次に向かうべき区画を得るために迷路探索アルゴリズムを実行することができる．ロボットの機体中心が目標区画の中心に移動する前に，壁情報の記録と探索アルゴリズムが終了するならば，ロボットは目標区画への到着したのち，瞬時に次の目標区画へと進行することができ，ロボットが迷路を探索している時間を短縮することができる．すなわち，リアクティブ動作と heavy-task 実行とを協調させることで，ロボットの探索性能が改善できる．以降では，リアクティブ動作と heavy-task 実行を交互に繰り返す探索を標準探索，リアクティブ動作と heavy-task 実行を並行に実行し探索性能を改善したものを改善探索と呼ぶ．

3.3.2 Emfrp での例題の記述

前述したロボットの制御プログラムを Emfrp で記述することを考える．ロボットの機体中心を PID 制御によって目標座標へ移動させる Emfrp モジュール `MovePID`を図 6 に示す．このモジュールは

```
# Emfrp による迷路探索ロボット
module MazeRunnerEmfrp
in sn : Int, se : Int, ss : Int, sw : Int, # 赤外線センサー
   enc1: Int, enc2: Int, enc3: Int, # 車輪の前回からの相対回転数
   time(0) : Float, # プログラム開始からの起動時間 (秒)
   to_u : Int, to_v : Int # 目標区画のインデックス
out achieved : Bool, # ゴールに到着したかどうか
    duty1 : Int, duty2 : Int, duty3 : Int, # 車輪に出力するduty 比
    reached_target : Bool, # 目標座標に到着したかどうか
    wall_n : Bool, wall_e : Bool, # 壁情報 (北 , 東)
    wall_s : Bool, wall_w : Bool  # 壁情報 (南 , 西)
use Std

data GOAL_U = ... # ゴール区画のX 座標
data GOAL_V = ... # ゴール区画のY 座標
data SECTION_WIDTH = ... # 区画の大きさ
data THR_WALL = ... # 壁があるかどうかの閾値
data THR_REACHED_TARGET = ... # 目標地点に到達したとみなす距離

# 前回のイテレーションにかかった時間
node duration = time - time@last

# 位置の制御
newnode target_dist, d1, d2, d3 =
  MovePID(to_u * SECTION_WIDTH, to_v * SECTION_WIDTH, enc1, enc2, enc3, duration)

node duty1 = if reached_target then 0 else d1
node duty2 = if reached_target then 0 else d2
node duty3 = if reached_target then 0 else d3

# 目標区画へ到達したかどうか
node reached_target = target_dist < THR_REACHED_TARGET

# 壁情報
node wall_n = sn > THR_WALL
node wall_e = se > THR_WALL
node wall_s = ss > THR_WALL
node wall_w = sw > THR_WALL

# ゴールに到達したかどうか
node achieved = (to_u == GOAL_U && to_v == GOAL_V && reached_target)
```

図 7. Emfrp による迷路探索ロボット制御モジュール

入力として目標座標 (tar_x，tar_y)，それぞれのモーターの回転数 (enc1，enc2，enc3)，前回のイテレーションからの経過時間 (duration) をとり，出力として目標座標との直線距離 (distance)，モーターそれぞれの出力比率 (duty1，duty2，duty3) を計算する．モジュールの内部ノードとして機体の現在位置 (cur_x，cur_y) を保持し，ロータリーエンコーダーの値を使って更新し続けている．さらに，機体座標について PID 制御を行い目標座標に近づくための制御量 (モーターの出力比率) を計算している．より正確にはロボットの回転についても制御の必要があるが，ここではロボットは回転しないものとして扱う．考慮する場合もロータリーエンコーダーやジャイロセンサーの情報から機体の回転についての PID 制御をすれば良い．

標準探索を行うロボットの Emfrp モジュール MazeRunnerEmfrpを図 7 に示す．さらに，MazeRunnerEmfrp モジュールのための C 言語による入力関数と出力関数の疑似コードを図 8 に示す．

MazeRunnerEmfrpモジュールは入力として，4 方向の赤外線センサーの値 (sn，se，ss，sw)，それぞれのモーターの回転数 (enc1，enc2，enc3)，プログラム開始からの時間 (time)，目標区画のインデックス (to_u，to_v) をとる．24 行目にて，前述の MovePIDモジュールをサブモジュールとして内部で使用することで，ロボットの機体中心が入力された目標区画 (to_u，to_v) の中心に移動するようなモーター出力 (d1，d2，d3) が計算される．27–29 行目は，目標区画に到着した場合には，それ

```
1  int next_u = 0; // 次の目標区画のX座標
2  int next_v = 1; // 次の目標区画のY座標
3  bool has_next = false; // 次の目標区画が計算済みであるかどうか
4
5  void Input(
6      int* sn, int* se, int* ss, int* sw,
7      int* enc1, int* enc2, int* enc3,
8      float* time, int* to_u, int* to_v){
9      get_sensors(sn, se, ss, sw); // 壁センサーの値を取得
10     get_encoders(enc1, enc2, enc3); // それぞれのモーターの回転数を取得
11     get_time(time); // 起動時間を取得
12     if(has_next){
13         *to_u = next_u; *to_v = next_v; // 目標区画を設定
14         has_next = false;
15     }
16 }
17
18 void Output(
19     bool achieved, int duty1, int duty2, int duty3,
20     bool reached_target,
21     bool wall_n, bool wall_e, bool wall_s, bool wall_w){
22     if(achieved) set_led(); // ゴールに着いたらLEDをつける
23     set_motor(duty1, duty2, duty3); // モーターへ出力
24     if(reached_target){ // 目標区画に到着したら壁情報を記録して，次の目標区画を計算する
25         // それぞれheavy-task
26         set_wall(next_u, next_v, wall_n, wall_e, wall_s, wall_w);
27         calc_next(&next_u, &next_v);
28         has_next = true;
29     }
30 }
```

図 8. MazeRunnerEmfrpモジュールのための入出力関数 (疑似コード)

ぞれのモーターへの出力を 0 にすることでロボットを停止させることを意図した記述である．次に，図 8 では，グローバル変数 (next_u，next_v，has_next) を用意することで，Output関数で計算された「次に向かうべき区画」を Input 関数へフィードバックしている．MazeRunnerEmfrpモジュールの出力ノード reached_targetが真の場合には，ロボットは目標区画へ到着し停止している．そこで，図 8，24 行目 (Output関数内) にてその状態を検知し，その時の壁情報を登録 (set_wall関数)，次に進むべき区画を計算 (calc_next関数) する．計算の間は Emfrp のイテレーション処理は停止している．計算が終わると，Output関数は終了するため，Emfrp のイテレーションが再び起動する．次のイテレーションにて，Input関数が呼び出された時，グローバル変数 has_nextは真であるから目標座標を新しく設定する．すると現在位置と新しい目標区画は離れているので，reached_targetノードは偽となり，目標区画に近づくためのイテレーションが実行されることになる．

以上の挙動から，出力関数内で heavy-task を行い，結果を入力関数へとフィードバックすることで，標準探索を実装することができた．しかし，前述の通り，出力関数内で heavy-task を実行している最中は Emfrp のイテレーションは停止してしまうため，機体の移動の他にリアクティブな動作がある場合には，その動作の応答性が悪くなってしまう．また，C 言語で記述された heavy-task によって出力関数から入力関数へのフィードバックが発生するため，必然的に出力ノードから入力ノードへの依存関係が生じる．しかし，このような暗黙的な依存関係，および heavy-task の対象となっているデータ構造が Emfrp プログラム中で明示されることはない．

3.3.3 Emfrp と RTOS 等との協調

前項の実装では，出力関数内で heavy-task を実行していたため，イテレーションが停止するという問題があった．そのため，Emfrp によるナイーヴな実装では改善探索を実装することは難しいと考えられる．そこで，RTOS 等の非同期タスク実行をサポートするライブラリや実行基盤を使用し，

Emfrp のイテレーションと並行に heavy-task 実行し，改善探索を実装する場合を考える．その場合，イテレーション中に heavy-task 実行が挟まるため，イテレーション間隔が長くなると考えられるが，長時間停止することはないため，応答性に問題はないと考えられ，目標地点に移動しながら，次の目標区画を計算することが可能である．しかし，heavy-task 実行が全て C 言語で行われてしまうため，入出力ノード間の暗黙的な依存関係や heavy-task の対象となっているデータ構造が Emfrp プログラム中で明示されることはない問題が解決できていない．また，Emfrp のイテレーションと RTOS のタスク間でデータのやりとりをする必要があり，一般的な並行プログラミング技法を活用することになると考えられる．これにより，従来の並行プログラミング特有の難しさが顕在化してしまい，Emfrp を使用する利点が損なわれてしまう．

4 提案手法

リアクティブな動作と heavy-task 実行を両立しつつ前節で述べた問題を解決するために，Emfrp に対して非同期タスク処理機構を導入する．導入に際して，言語拡張とともに言語ランタイムの変更が必要となる．本節では，非同期タスク処理機構を導入した Emfrp で記述した迷路探索ロボットの例を通し，言語拡張，ランタイム拡張について説明する．

4.1 言語拡張

言語拡張の説明のため，非同期タスク処理機構を導入した Emfrp で記述した迷路探索ロボットの例を図 9 に示す．さらに，このモジュールにインポートされているマテリアルを図 10 に示す．以降では，heavy-task を表す非同期タスクを単にタスクと呼び，そのタスクの対象となるデータ構造をタスクリソースと呼ぶ．

4.1.1 拡張の概要

非同期タスク処理機構の導入にあたり新たに Emfrp に新たに追加される言語要素としては，タスクとそのタスクの対象となるタスクリソースを定義する構文 (図 10，20–29 行目)，モジュール間でタスクリソースを受け渡す機構 (図 9，4 行目),「計算状態」を表す `Future`型，タスクと `Future`型ノードとを束縛する構文 (図 9，26–28 行目，31–33 行目) の 4 つである．

4.1.2 タスクリソースとタスクの定義

タスクとタスクリソースは図 10，20–29 行目のように定義する．定義の先頭はタスクリソースの名前であり，型のように扱われる．続いて g`:(`$x_1 : \bar{\tau}$`,...,`$x_n : \bar{\tau}$`)->(`$y_1 : \bar{\tau}$`,...,`$y_m : \bar{\tau}$`)/`pの形でタスクを宣言する．g はタスク名，x_i は入力の名前，y_i は出力の名前，$\bar{\tau}$ は `Future`型 (後述) でない型，p は **`read`**または **`write`**のいずれかである．p は，複数タスクが同一のタスクリソースを非同期的に扱うことで起こりうるデータ競合を避けるためのコンパイラ及びランタイムへのヒントである．この例では，タスクリソース `MazeGraph`のインスタンスに対し，書き込み操作するタスクは `RegisterSection`，読み込み操作するタスクは `CalcNextSection`ということを意味する．

図 9，4 行目では，モジュールの引数としてタスクリソースのインスタンスが受け渡されている．受け取ったインスタンスはサブモジュールへと受け渡すことができ，サブモジュール内でそのインスタンスに対する **`tasknode`**(後述) を定義できる．`MazeRunner`モジュールのようにトップレベルモジュールの場合には，プログラム起動時にリソースのインスタンスが受け渡される (4.2 節にて解説する)．

```
# MazeRunner.mfrp
# 拡張Emfrpによる迷路探索ロボット
module MazeRunner
in mg : resource MazeGraph, # heavy-taskリソースの指定
   sn : Int, se : Int, ss : Int, sw : Int, # 赤外線センサー
   enc1: Int, enc2: Int, enc3: Int, # 車輪の前回からの相対回転数
   time(0) : Float, # プログラム開始からの起動時間(秒)
out achieved : Bool, # ゴールに到着したかどうか
    duty1 : Int, duty2 : Int, duty3 : Int # 車輪に出力するduty比
use Std, Resources

# 前回のイテレーションにかかった時間
node duration = time - time@last

# 位置の制御
newnode target_dist, duty1, duty2, duty3 =
  MovePID(to_u * SECTION_WIDTH, to_v * SECTION_WIDTH, enc1, enc2, enc3, duration)

# 機体中心が目標区画に十分に入ったかどうか
node init[False] in_target_section = target_dist < SECTION_WIDTH * 0.4

# 目標区画へ到達したかどうか
node reached_target = target_dist < THR_REACHED_TARGET

# 区画情報の登録
tasknode finish_register : Future[Unit] =
  mg.RegisterSection(to_u, to_v, wall_n, wall_e, wall_s, wall_w)
    trigger (!in_target_section@last && in_target_section) # 区画の中心にある程度近づいた瞬間(立ち上がりエッジ)

# 探索結果(既にゴールに着いていたら移動しない(ゴール座標を返し続ける))
tasknode (next_u : Future[Int], next_v : Future[Int]) =
  mg.CalcNextSection(to_u, to_v, GOAL_U, GOAL_V)
    trigger wait(finish_register) # mg.RegisterSectionが終わったら続けて実行する

# 目標区画への侵入方向(機体がどの方向へ進もうとしているか)
node move_dir = (to_u - from_u, to_v - from_v) of:
  (0, 0) -> Stop, # 停止
  (0, 1) -> StoN, # ↑
  (0, -1) -> NtoS, # ↓
  (1, 0) -> WtoE, # →
  (-1, 0) -> EtoW, # ←
  _ -> Stop # unreachable

# 壁情報(区画入り口にて有効な値)
node wall_n = move_dir of:
  Stop -> sn > THR_WALL_SHORT,
  StoN -> sn > THR_WALL_LONG,
  NtoS -> False, #「↓方向から区画に侵入できる」=「北壁はない」
  WtoE -> sn > THR_WALL_SHORT,
  EtoW -> sn > THR_WALL_SHORT
node wall_e = ...
node wall_s = ...
node wall_w = ...

# 目標区画の更新(最初は必ず前進する)
node init[(0, 0, 0, 1, None)] (from_u, from_v, to_u, to_v, tmp) =
  (reached_target, next_u, next_v, tmp@last) of:
    (True, Ready(nu), Ready(nv), _, _) -> (to_u@last, to_v@last, nu, nv, None)
    (False, Ready(nu), Ready(nv), _, _) -> (from_u@last, from_v@last, to_u@last, to_v@last, Some((nu, nv)))
    (True, _, _, Some((nu, nv))) -> (to_u@last, to_v@last, nu, nv, None)
    _ -> (from_u@last, from_v@last, to_u@last, to_v@last, tmp@last)

# ゴールに到達したかどうか
node achieved = move_dir of:
  Stop -> to_u == GOAL_U && to_v == GOAL_V && reached_target,
  _ -> False
```

図 9. 拡張 Emfrp による迷路探索ロボット制御モジュール

```
# Resources.mfrp
material Resources;

# 移動方向を表す型
type MoveDir = Stop | StoN | NtoS | WtoE | EtoW
data GOAL_U = ... # ゴール区画のX 座標
data GOAL_V = ... # ゴール区画のY 座標
data SECTION_WIDTH = ... # 区画の大きさ
data THR_WALL_SHORT = ... # 壁があるかどうかの閾値 (近距離)
data THR_WALL_LONG = ... # 壁があるかどうかの閾値 (長距離)
data THR_REACHED_TARGET = ... # 目標地点に到達したとみなす距離

# heavy-task が完了するのを待つ関数
func wait(s : Future[A]) = s of:
  Ready(_) -> True,    # タスクが完了した直後のイテレーションのみReady になる
  Pending -> False,    # タスクが発行されたが , 実行は完了していない状態
  NotStarted -> False # タスクが発行されていない状態

# タスクリソースとそのリソースをあつかうタスク (heavy-task)の定義
resource MazeGraph {
  # 区画 (u,v)の壁情報を MazeGraph のインスタンスに記録するタスク
  RegisterSection :
    (u : Int, v : Int, n : Bool, e : Bool, s : Bool, w : Bool) -> (h : Unit) / write

  # 区画 (u,v)からゴールに行くために , 次に行くべき区画 (next_u,next_v)を計算するタスク
  CalcNextSection :
    (u : Int, v : Int, goal_u : Int, goal_v : Int)
      -> (next_u : Int, next_v : Int) / read
}
```

図 10. MazeRunnerモジュールで使用されるマテリアルファイル

4.1.3 Future 型と tasknode 定義

タスクを非同期に実行するため，そのタスクの結果を表すノードは「まだ計算が終わっていない」状態を表現する必要がある．このような場合には，future と呼ばれるデータ型を導入し,「まだ計算が終わっていない」式に future 型をつけることが並行プログラミングの文脈では一般的である．そこで，本研究では `type Future[A] = Ready(A) | Pending | NotStarted`として定義される Future型を導入した．この値は Emfrp 上でのヴァリアント型と全く同様にパターンマッチによる分解ができる．Future型のノードが Ready(_)の値を持つのは，そのノードに紐づいたタスクが完了した直後のイテレーションの間だけである．Pendingはタスクは実行されている (実行待ちになっている) が，完了しておらず結果が得られていない状態を表す．NotStartedはタスクが実行待ちになっていない状態を表す．

Future型のノードは **tasknode** $(z_1$:Future[$\bar{\tau}$],...,z_m:Future[$\bar{\tau}$]) = $r.g(e_1,\ldots,e_n)$ **trigger** e の構文により定義される．ここで，z_i はノード名，r はタスクリソースのインスタンス，e_i はタスクに渡す入力式，e はタスクの発行条件式である．タスクが実行待ちになっていないかつ発行条件式が真である場合，そのタスクは実行待ちになる．実行待ちになる際には，タスクの入力式 (時変値) はその時点での値が保存され，タスク実行中は保存された値が使用される．また，$z_1,\ldots,z_m$ について，タスク g が完了した時に全て同時に値が Ready(_)となる．

tasknodeによるノード定義にも通常のノードと同様に依存関係の循環検査が行われる．検査時にはタスク入力式とタスク発行条件式中に参照したノードに依存関係があるとみなされる．また，Future型のノードに関しても通常のノードと同様に@lastによる直前値参照が可能である．

最後に，**tasknode**定義の制限について説明する．プログラム全体を通して，あるタスクについての **tasknode**定義はそれぞれのタスクリソースインスタンスごとに 1 つずつしか存在できない．つまり，MazeRunnerモジュールについて，タスクリソースインスタンス mgを利用する RegisterSectionタ

```c
// Resource.c
struct MazeGraph {
    /* Fill Your Code */
};

#define STACKSIZE_RegisterSection 2000
void RegisterSection(
    struct MazeGraph* w_res,
    int u, int v, int n, int e, int s, int w, int* h) {
    /* Fill Your Code */
}

#define STACKSIZE_CalcNextSection 2000
void CalcNextSection(
    const struct MazeGraph* r_res,
    int cur_u, int cur_v, int goal_u, int goal_v,
    int* next_u, int* next_v) {
    /* Fill Your Code */
    /*  r_res に破壊的変更を加えてないけない  */
}
```

図 11. タスクリソースとタスクのコンパイル結果 (一部)

スクについての **tasknode**定義は 26 行目で行われているため，同様の **tasknode**定義は禁止される．この制限により，プログラムの実行中に実行キューに存在しうるタスクの最大数を静的に決定することができる．

4.2 言語ランタイムの実装方針

本研究では，小規模組込み環境でのプログラム実行を想定している．そのため，以降ではシングルコアマイコン向けの言語ランタイムの実装方針を説明する．

4.2.1 タスクリソースとタスク

図 10 にて定義したタスクリソースとタスクはコンパイラによって，図 11 に示すテンプレートコードに変換される．拡張 Emfrp 上でのタスクリソースは，C 言語では同名の構造体へと変換され，その内部構造はユーザーが補完する．タスクは C 言語での同名の関数に変換され，第一引数にタスクリソースインスタンスを表す構造体のポインタを受け取る．それぞれの関数内では，タスク定義時に指定した **read**または **write**に応じて，タスクリソースインスタンスを扱うことが想定されている．タスクにはそれぞれ個別のスタック領域が静的に確保され，タスク実行時に割り当てられる．その大きさは `STACKSIZE_XXX`定数によって指定される．

次にトップレベルモジュールから生成されるメイン関数と入出力関数のテンプレートコードを図 12 に示す．既存の Emfrp のテンプレートコードとの違いは，タスクリソースインスタンスである構造体がグローバル変数 `mg`として定義されていることである．また，トップレベルモジュールの入力として指定されたタスクリソースは，`ActivateMazeRunner`関数の引数として渡されるようにコンパイルされる．ユーザーは `mg`の初期化処理を行ったのち，それを入力として `ActivateMazeRunner`関数を呼び出すことで，Emfrp プログラムを開始する．

4.2.2 イテレーション

2.5 節にて説明したイテレーションに対し，非同期タスク実行のためのフェーズが追加される．具体的には，出力関数の呼び出しと直前値更新処理の間に非同期タスク実行が行われる．そのため，拡張 Emfrp のイテレーションは既存の Emfrp のものよりも長い実行時間であることが予想され，応答性が劣化する．イテレーション中のタスク実行は，コンパイル時にユーザーの指定する時間のみ

```c
// MazeRunnerMain.c

// task resource instance
struct MazeGraph mg;

void Input(...){
    /* Fill Your Code */
}

void Output(...){
    /* Fill Your Code */
}

int main(void){
    /* センサーなどの初期化 */
    /* タスクリソースインスタンスmg の初期化 */
    ActivateMazeRunner(&mg);
    return 0;
}
```

図 12. メイン関数と入出力関数のテンプレートコード (一部)

行うことを想定している．それゆえ，応答性の劣化が許容できる範囲になるようにユーザーがタスク実行時間を定めることができる．

4.2.3 タスク実行基盤

本研究では，非同期タスク実行基盤を設計する際に，1. タスク間で共有しているタスクリソースの一貫性が保たれること，2. マイコンのような小規模環境で動作すること，3. ランタイムが動的なメモリ確保を必要としないことを方針とした．

方針 1 を満たすために，タスク定義での `read`と `write`をヒントとして利用する．タスクリソースの一貫性を保つという観点から，同一のリソースに対するタスクは`read`タスク同士ならば複数個並行に実行されていても問題はない．しかし `write`タスクの場合には，そのタスクが完了するまで他の同一のリソースに対するタスクは実行すべきではない．

そこで，タスク実行基盤として言語ランタイムは，1 つの実行リストとタスクリソースごとの実行待ちキューを持ち，タスクの実行を制御する．実行の途中であるタスクは実行リストに含まれている．発行はされているが実行できない状態のタスクは実行待ちキューにエンキューされる．発行はされているが実行できない状態とは，タスクリソース A を対象とした `write`タスクが実行リストに含まれている場合に A を対象とした別タスクを発行した場合や，タスクリソース A を対象とした `read`タスクが実行リストに含まれている場合に A を対象とした `write`タスクを発行した場合である．ノード更新処理中に，タスクの発行条件が満たされた場合には，そのタスクを発行し実行リストまたは実行待ちキューに加える．

アルゴリズム 1 に，以上で説明したタスク発行処理と，`Future`型のノードの更新処理を行う関数の疑似コードを示す．この関数はイテレーション中のノード更新処理の際に依存関係の順に従って呼び出される．アルゴリズム中，タスクリソースインスタンス r を扱うタスク x を表すオブジェクトを x^r と表記する．実行基盤におけるタスクの状態は x^r.state と表され，発行されていない状態 (NotStarted)，実行待ち (Pending)，完了直後 (Ready) のいずれかを取る．また，E はタスクの実行リスト，W_r はタスクリソースインスタンス r の実行待ちキューを表す．

非同期タスク実行はラウンドロビン方式で，1 イテレーションにつき 1 タスクが行われる．それぞれのイテレーションごとに実行リストにあるタスクを選び実行する．タスク実行と休止にはオペレーティングシステムでのコンテキストスイッチの技法とタイマ割り込みを利用する．また，タスク実行の際にはタスクごとに用意されたメモリ領域をスタック領域として使用する．イテレーション中のタスク実行フェーズでは，コンパイル時に設定された最長タスク実行時間までタスク実行が

アルゴリズム 1 リソースインスタンス r のタスク $x(x^r)$ の発行と future ノードの更新

```
1: function MUTUAL_EXCLUSION(x^r, E)                          ▷ タスクの相互排他条件
2:     return (x^r は read タスク ∧ E 中に r の write タスクが含まれていない) ∨ (x^r は write タスク ∧
   E 中に r のタスクが含まれていない)
3: end function
4:             ▷ E は実行リスト (キュー)，W_r はタスクリソースインスタンス r の実行待ちキューを表す
5: function UPDATE_AND_ISSUE_r_x( )                           ▷ タスクの発行と関連ノードの更新
6:     if x^r.state = Pending then
7:         ノードそれぞれに Pendingを出力
8:         return
9:     end if
10:    if x^r.state = NotStarted then
11:        ノードそれぞれに NotStartedを出力
12:    else if x^r.state = Ready then
13:        ノードそれぞれに Ready(x^r の計算結果)を出力
14:        x^r.state ← NotStarted
15:    end if
16:    if x^r の発行条件式が真 then
17:        x^r.state ← Pending
18:        x^r への入力値を計算，保存
19:        if W_r = ∅ ∧ MUTUAL_EXCLUSION(x^r, E) then
20:            E.enqueue(x^r)
21:        else
22:            W_r.enqueue(x^r)
23:        end if
24:    end if
25: end function
```

可能である．これを超える場合には，タイマ割り込みによってタスク実行が休止する．タスクが完了した場合には，実行待ちキューを探索し，新しい実行待ちタスクを実行リストへと加える．この方式はタイマ割り込みがあれば実現でき，複雑なスケジューラは必要ないため，方針 2 が満たされる．アルゴリズム 2 に非同期タスク実行を行う関数の疑似コードを示す．この関数はイテレーションごとに 1 回呼び出され，実行リスト内のタスクを断続的に実行する．

4.1.3 節で説明した **tasknode**定義の制限のため，タスクリソースのインスタンス数やタスクの数は静的に決定することができる．その情報をもとに実行リストや実行待ちキューの長さをコンパイル時に決定でき，方針 3 を満たすことができる．

注意事項として，非同期実行されるタスクは C 言語で記述された関数である．そのため Emfrp によるメモリ安全性などの保証はない．タスク実行のために用意されたメモリではスタック領域として不足する場合があり，プログラム実行中の予期せぬバグとなりうる．これを防ぐためには StackAnalyzer[4]等の外部ツールによって安全性を保証する必要がある．

4.3 例題の動作

以上の拡張が導入された Emfrp を用いて記述された図 9 の MazeRunnerモジュールの動作を解説する．まず，4 行目にてタスクリソースとして MazeGraphが使用されることが明示されている．他の入出力ノードに関しては，拡張前の MazeRunnerEmfrpモジュールとほぼ同様である．ただし，壁情報の記録や目標区画の計算がモジュール内で完結するようになったため，入出力ノードには含まれていない．16 行目にて MovePIDモジュールをサブモジュールとして展開している．したがって，目標区画の座標 (to_u, to_v) の変更と同時に機体が移動する．

[4] https://www.absint.com/stackanalyzer/index.htm

アルゴリズム 2 非同期タスク実行

```
 1: function EXECUTE_TASK( )                                          ▷ 非同期タスク実行
 2:     if E = ∅ then
 3:         return
 4:     end if
 5:     x^r ← E.dequeue()
 6:     x^r へコンテキストスイッチして N 秒間実行，元のコンテキストへ戻る
 7:                                                  ▷ N はコンパイル時に指定される定数
 8:     if x^r の実行が完了した then
 9:         x^r.state ← Ready
10:         x^r の出力結果を保存
11:         while W_r ≠ ∅ ∧ MUTUAL_EXCLUSION(W_r.peek(), E) do
12:             y^r ← W_r.dequeue()
13:             E.enqueue(y^r)
14:         end while
15:     else
16:         E.enqueue(x^r)
17:     end if
18: end function
```

20 行目の in_terget_setionノードは，機体中心が目標区画の中に十分入った時に真になる．このノードの立ち上がりエッジを検出して，26 行目の壁情報記録タスク (RegisterSection) を発行している．このタスクは完了すると Unit型の値を返す．図 10 の 14 行目にて定義された wait関数により，Future型のノードの計算完了を立ち上がりエッジとして検出することができる．31–33 行目では，この関数を利用して壁情報の記録が終わったのちに迷路探索タスク (CalcNextSection) を発行している．すなわち，機体が目標区画にある程度近づいた状態になると，目標区画の壁情報が記録され，次に目標となるべき区画の探索が始まっている．また，その探索中も MovePIDモジュールは動作を続けているため，リアクティブな動作が停止することもない．CalcNextSectionタスクの完了を検知するのはノード定義式のパターンマッチを行なっている 58 行目と 59 行目である．58 行目は機体が目標区画に到着してからタスクが完了したことを意味する．この場合には新しい目標区画をすぐに (to_u，to_v) に反映させる．59 行目は機体が目標区画に到着する前にタスクが完了したことを意味する．この場合には新しい目標区画を一旦ペアのオプショナル型である tmpノードに退避する．その後，目標区画に到着した瞬間には 60 行目にマッチするため，退避してあった次の目標区画の情報が (to_u，to_v) に反映される．

新しい目標区画が設定されると in_terget_setionノードは偽になる．機体が新しい目標区画に十分に近づくと in_terget_setionノードは真になり，立ち上がりエッジが発生する．したがって以上に述べた動作が機体がゴールに到着するまで繰り返されることになる．

以上より，MazeRunnerモジュールでは改善探索が実装できている．また，このプログラムでは，heavy-task の対象となっているデータ構造が明示されるわけではないものの，タスクリソースインスタンスを介して，どのリソース (データ構造) が操作されているのかが表されている．3.3.3 節で問題となっていた入出力ノード間の暗黙的な依存関係や操作中のリソースが明示されない問題が解決され，MazeRunnerモジュールは比較的見通しの良いプログラムになっていると言える．

5 関連研究

future パターンは並行プログラミングにおけるデザインパターンの一つであり，非同期処理の結果の取得を実際にその値が必要になるまで遅延させる手法である．Java，JavaScript，Scala，Rust など様々な言語で実装されている．リモートプロシージャコールなど，結果の取得に時間のかかる

操作を別のスレッドで非同期に実行し，結果を待つ間に別の処理を行うパイプラインを構成するために用いられる．本研究においても，まさに heavy-task 実行は非同期処理であり，発行されたタスクの計算状態を管理，プログラム中で利用する必要性から導入に至った．

リアクティブプログラミング [2] では，リアクティブな動作を時変値間の依存関係グラフにおけるデータフローとして表現する．そのため，本研究で heavy-task と呼んでいる動作のような，データフローとして扱いにくい動作をどう表現するかは言語設計上の問題となる．Van den Vonder らは，手続き型言語をベースとしたリアクティブプログラミング言語（例えば REScala[14], ReactJS[8], RxJS[17]）を対象としてこの問題を分析し，リアクティブな動作の記述と手続的な動作の記述を分離してモジュール化する手法である Actor-Reactor モデルを提案している [18]．このモデルでは，時間のかかる処理（heavy-task）や副作用のある処理等を手続き的に記述されるアクター [1] としてモジュール化し，データフローグラフとして表現されるリアクティブな動作の記述（リアクター）と分離する．

Actor-Reactor モデルにおけるアクターとリアクター間の協調は，リアクターの出力ノードからアクターを経由して当該リアクターの入力ノードへフィードバックする形で行われる．そのため，第 3.3.2 節で議論している，出力ノードと入力ノードの暗黙的な依存関係によってプログラムの見通しが悪くなる問題がある．本研究では，`tasknode`定義と `Future`型の導入によってこの問題を解決している．

Hailstorm [15] は Arrowized FRP [10, 9] の計算モデルに影響を受けた IoT アプリケーション向けの関数型言語である．構文も Arrowized FRP の影響を強く受け，`&&&`演算子や`>>>`演算子を使用してシグナル関数 (Signal funnction) を合成しながらプログラムを記述する．プログラムの記法は異なるものの，その実行モデルは Emfrp のものと類似している．そのため，計算処理に時間のかかるシグナル関数を合成する場合には Emfrp と同様に応答性が悪化すると考えられる．これを改善するために本研究のアプローチも有用だと考える．

6　まとめ

小規模組込みシステム向け FRP 言語 Emfrp に対し，リアクティブな動作と比較的時間のかかるタスクとを協調可能にする非同期タスク実行基盤を設計した．言語拡張としては，非同期タスクを発行しその結果を得るために新たに `tasknode`定義と `Future`型の導入を行なった．ランタイム拡張としては，既存の時変値更新処理の途中にタスク実行フェーズを挿入し，小規模環境においても動作可能な非同期実行方式を提案した．

本提案手法では，タスクが操作するリソースを全てまとめてタスクリソースとして扱う．タスクリソースをより細かいリソースに分割し，タスク間でその合流や共有を行うことで，より粒度の細かい排他制御やリソース管理が可能である．これを実現するための記述方式や実行基盤の設計及びタスク実行の優先度設定については将来課題とする．

今後の方針としては，本研究の提案手法を実装しシステムの応答性や heavy-task 実行についての時間的オーバーヘッド，非同期タスク実行のために必要なランタイムメモリの空間的オーバーヘッドを計測予定である．また，意味論を形式的に定義することも予定している．

謝辞

本研究の一部は JSPS 科研費 21K11822 および 19K20245 の助成を受けている．本研究は JST 次世代研究者挑戦的研究プログラム JPMJSP2106 の支援を受けている．

参考文献

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, Vol. 45, No. 4, pp. 52:1–52:34, 2013.

[3] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pp. 7–18, New York, NY, USA, 2003. ACM.

[4] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pp. 411–422. ACM, 2013.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, pp. 263–273. ACM, 1997.

[6] Caleb Helbling and Samuel Z Guyer. Juniper: A functional reactive programming language for the Arduino. In *4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*, pp. 8–16. ACM, Sep. 2016.

[7] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, Vol. 2638 of *Lecture Notes in Computer Science*, pp. 159–187. Springer-Verlag, 2003.

[8] Facebook Inc. ReactJS: A Javascript library for building user interfaces. https://reactjs.org, Accessed Jan. 2022.

[9] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. *SIGPLAN Not.*, Vol. 44, No. 9, pp. 35–46, aug 2009.

[10] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, p. 51–64, New York, NY, USA, 2002. ACM.

[11] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *4th International Conrefernce on Principles and Practice of Declarative Programming (PPDP 2002)*, pp. 168–179. ACM, 2002.

[12] Yoshitaka Sakurai, Sosuke Moriguchi, and Takuo Watanabe. Functional reactive programming for embedded systems with GPGPUs. In *10th International Conference on Software and Computer Applications (ICSCA '21)*, pp. 75–80. ACM, Feb. 2021.

[13] Yoshitaka Sakurai and Takuo Watanabe. Towards a statically scheduled parallel execution of an FRP language for embedded systems. In *6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2019)*, pp. 11–20. ACM, Oct. 2019.

[14] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *13th International Conference on Modularity (Modularity 2014)*, pp. 25–36. ACM, 2014.

[15] Abhiroop Sarkar and Mary Sheeran. Hailstorm: A statically-typed, purely functional language for iot applications. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20, New York, NY, USA, 2020. ACM.

[16] Kensuke Sawada and Takuo Watanabe. Emfrp: A functional reactive programming language for small-scale embedded systems. In *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*, pp. 36–44. ACM, Mar. 2016.

[17] RxJS Team. Rxjs: Reactive extensions library for JavaScript. https://rxjs.dev, Accessed Jan. 2022.

[18] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. Tackling the awkward squad for reactive programming: The actor-reactor model. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Vol. 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 19:1–19:29. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Nov. 2020.

[19] Takuo Watanabe. A simple context-oriented programming extension to an FRP language for small-scale embedded systems. In *10th International Workshop on Context-Oriented Programming (COP 2018)*, pp. 23–30. ACM, Jul. 2018.

[20] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. A functional reactive programming language for small-scale embedded systems with recursive data types. *Journal of Information Processing*, Vol. 29, pp. 685–706, Oct. 2021.