論文 / 著書情報
Article / Book Information

題目(和文)	リソースアウェアなリアクティブプログラミング言語
Title(English)	
著者(和文)	横山陽彦
Author(English)	Akihiko Yokoyama
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12744号, 授与年月日:2024年3月26日, 学位の種別:課程博士, 審査員:渡部 卓雄,権藤 克彦,西崎 真也,DEFAGO XAVIER,林 晋平
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12744号, Conferred date:2024/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

博士論文

リソースアウェアな
リアクティブプログラミング言語

横山 陽彦

東京工業大学
情報理工学院
情報工学コース

指導教員　渡部 卓雄

2024 年 3 月

概要

近年，サイバーフィジカルシステム (CPS) が発達してきている．CPS とは計算プロセスと物理プロセスの統合である．エッジデバイスと呼ばれる組込み機器が計算の世界と実世界の境界となる．CPS では，エッジデバイスを介した計算と物理プロセス間のフィードバックによって，それらの相互作用が行われる．CPS の重要な特徴には，「高い確実性」と「迅速な応答」がふくまれる．エッジデバイスは，この二つの特徴に従うことが要求される．すなわち，不確実な環境のもとでも安定的に動作する組込みシステムの開発が重要である．

エッジデバイスのように，非同期的な入力に対して内部状態を変更させながら反応するシステムのことをリアクティブシステム (RS) という．関数リアクティブプログラミング (FRP) は，RS を簡潔に記述するためのプログラミングパラダイムである．FRP では，ユーザーは時変値 (時間とともに自動的に変化する値) を使用して，入力から出力までのデータフローを記述することで，RS をシンプルに構築できる．FRP と組込みシステムの相性の良さから，組込みシステム向け FRP 言語がいくつか提案されてきた．小規模組込みシステム向け FRP 言語 Emfrp はそのような FRP 言語のうちの一つである．

しかしながら，Emfrp やその他の組込みシステム向け FRP 言語では計算リソースを詳細に扱う機構が不足していたため，エッジデバイスのように安定動作が要求される組込みシステムを記述することに難があった．そこで，本研究では組込みシステムにおいて重要な 3 つの計算リソースに注目し，それらのリソースにまつわる既存の FRP 言語の問題点の分析とその解決方法の提案を行なった．より具体的には，既存の組込みシステム向け FRP 言語を代表して Emfrp を対象に，メモリリソース，CPU リソース，電力リソースについての分析を行なった．それぞれのリソースごとに Emfrp 言語を拡張した新言語の提案によって，それらに対応した問題に対する解決策を提示した．これらの概要を以下に示す．

メモリリソースへのアプローチのために $\text{Emfrp}^{\text{BCT}}$ を提案した．Emfrp は一定のメモリスペースでの動作を目指すため，再帰的定義を禁止していた．この制限により，従来の関数プログラミングでの技法を Emfrp プログラム中で使用することができなかった．そのため，ユーザーは不自然なプログラム記述を強いられてしまう問題があった．本研究で提案する $\text{Emfrp}^{\text{BCT}}$ は，Emfrp にサイズ制限を持つ再帰データ型 (Bounded-Construction Types,BCT) を導入した拡張言語である．再帰的定義に関する制限の緩和とともに，BCT と時変値の親和性の向上を目的に fit 式等の構文の導入も行なった．$\text{Emfrp}^{\text{BCT}}$ では，ユーザーは再帰データ型を使用しつつも，プログラムの使用メモリ量の決定が可能である．$\text{Emfrp}^{\text{BCT}}$ の形式化をして健全性を示したのち，処理系を作成し各種オーバーヘッドの計測を行なった．

CPU リソースへのアプローチのために Emfrp^{HT} を提案した．時変値を直線的に更新する FRP 言語では，更新すべき時変値に 1 つでも時間のかかる計算 (Heavy-task) が含まれるとシステム全体の応答性が低下する．この現象は Reactive Thread Hijacking Problem(RTHP) と呼ばれ，問題視されてきた．本研究では，ロボット競技マイクロマウスでの例を提示しながら，現状の Emfrp での RTHP 回避手法における問題点を指摘し，その問題点を解決するために Emfrp^{HT} を提案した．Emfrp^{HT} は，Heavy-task を定義し時変値として扱う機構および Heavy-task を非同期実行するランタイムをもつ．Emfrp^{HT} では，リアクティブな振る舞いと Heavy-task による計算を協調動作させることができる．これは，システム全体の応答性の安定化に貢献する．

電力リソースへのアプローチのために EvEmfrp/S を提案した．Emfrp では，メモリフットプリントを小さくするために，ある種のポーリングによって時変値更新を連続的に行う．この仕組みでは，必要以上に高頻度な更新処理が行われ，消費電力が大きくなる問題があった．Emfrp の後続言語 EvEmfrp は，時変値更新処理を間欠実行にすることで，低電力化を図った．EvEmfrp では，時変値は更新頻度に関する注釈を持つ．これをヒントとしてコンパイラが更新間隔を適切にスケジューリングする．しかしながら，例えば組込みシステムで頻出する「非同期タイミングを起点として，一定時間後に処理を行う」動作を記述する際には，注釈の表現力の不足から十分な省電力効果が得られない可能性があった．本研究では，EvEmfrp のタイミング注釈を一新し，時変値更新頻度を動的に切り替える機構を持つ拡張言語 EvEmfrp/S を提案した．ソフトウェアによるボタンのノイズ除去の例を通して，EvEmfrp/S によるプログラムでは，従来の EvEmfrp よりも積極的にハードウェアスリープを活用することができ，省電力化が期待できることを示した．

これらの言語およびその機能の統合は今後の課題である．本研究の成果が，安定動作する組込みのための FRP 言語の構築へ向けた一歩となることを期待する．

目次

図目次

表目次

第 1 章

序論

1.1 はじめに

近年，**サイバーフィジカルシステム (CPS)** [19, 39, 63] の発達とともに，**安定動作する組込みシステム**の需要が殊更に高まっている．CPS とは，コンピューターによる計算と実世界の物理情報をネットワークによって統合し，それらの相互作用によって実現されるシステムの総称である．実世界の情報を観測するあるいは実世界に対して作用する主体は，エッジデバイスと呼ばれる組込み機器である．コンピューターによる世界と現実の世界の境界に配置されることから「エッジ」デバイスと名付けられたこれらの機器は，比較的小規模な計算機と複数のセンサーやアクチュエータの集まりから構成される．実世界の情報はエッジデバイスとそれに併設されたセンサーによって観測，計測され，ネットワークを介して別のコンピューターへとデータが送信，蓄積される．蓄積されたデータは例えばクラウド上の計算機によって処理が行われ，再びネットワークを介してエッジデバイスへと還元される．エッジデバイスはアクチュエータによって実世界に影響を与え，その結果が再びセンシングされるというフィードバックが繰り返される．

CPS の典型的なアプリケーションはスマート家電，ヘルスケア，工場の自動化，スマート農業，自動運転，電力網制御，災害対策など大小様々なものが挙げられるが，いずれもエッジデバイスを利用する．文献 [19] によれば，CPS の重要な特徴付けとして**高い確実性 (Highly Certainty)** と**迅速な応答 (Quick Response)** がある．例えば，ヘルスケアや災害対策のアプリケーションを想定すると，エッジデバイスにて体調や環境の異常を正確に検知することが必要である．さらに検出データをすぐに処理してネットワークに送信し，コンピューター世界での計算に従った対処法にすばやく反応してアクチュエータ等の起動を行うことが要求される．エッジデバイスはさまざまな形や用途での使用が想定される．広い農地にエッジデバイスを多数配置しセンサーネットワークを形成するために，コストダウンを図るために計算リソースを限った小規模なデバイスを使用することもある．ウェアラブルデバイスのように身につけることが前提のエッジデバイスでは，電池や環境発電によってシステムが駆動する．このような状況下においてもエッジデバイスには，確実性を持ったセンシングや素早い応答が期待される．すなわち，CPS の発展に欠かせないエッジデバイスの開発は，**不確実な環境のもとでも安定的に動作する組込みシステムの開発**であると言い換えることができる．

エッジデバイスのように，センサーなどの外部からの非同期的な入力に対して内部状態を変化させながら反応し続けるシステムを**リアクティブシステム**と呼ぶ．リアクティブシステムの典型例としては，グラフィカルユーザーインターフェース (GUI)，ビデオゲーム，家電やロボット等の組込みシステムなどがある．このようなシステムを一般的な手続き的なプログラミングスタイルで記述する際には，非同期的な入力に対応するためのプログラミング技法が用いられてきた．ポーリングによって入力待ちのループを構成する方法や，入力イベントに対する割り込みや関数コールバックの多用などである．これらのプログラミング技法の多用は，処理の流れ (コントロールフロー) や値の流れ (データフロー) を分

断するため，プログラムの可読性や保守性を損ねることが知られている [4]．特に関数コールバックが多用され，可読性が損なわれる現象はコールバック地獄 (Callback Hell) と呼ばれ問題とされてきた．

関数リアクティブプログラミング (Functional Reactive Programming，FRP) [14] は，時間とともに自動的に変化する値である**時変値 (time-varying value)** を宣言的に組み合わせることで，リアクティブシステムの記述を行うプログラミングパラダイムである．FRP によるプログラミングでは，ユーザーはシステムの入力から出力までを時変値の関係として記述する．これにより入力から出力までのデータフローが明示されることで，簡潔かつ保守性の高いコードの記述が可能である．前述のコールバック地獄は FRP によって解決することができる．FRP はその導入当初から関数型言語 Haskell 上のドメイン特化言語 (Domain Specific Language，DSL) として発展してきた [5, 31, 47]．また，組込みシステム記述との相性の良さから組込み環境や IoT 環境を想定した DSL がいくつか研究，開発され [25, 50, 58, 60, 62, 67, 75]，組込みシステム開発の一助となっている．

1.2 研究の目的

本研究の目的は，不確実な環境のもとでも安定に動作する組込みシステムの開発を支援するため，**計算リソースを適切に検査，制限，制御することのできる (リソースアウェアな) リアクティブプログラミング言語の構築を行う**ことである．

本研究の基礎とするリアクティブ言語は Emfrp [60] である．Emfrp は，小規模な組込み環境，具体的には数十 KB から数百 KB の RAM を搭載するマイクロコントローラを動作環境として想定しデザインされた，関数リアクティブプログラミング言語である．Emfrp は FRP の記法によるリアクティブシステムの簡潔な記述に長けており，2016 年の開発，発表から現在に至るまで複数の拡張が施されながら発展を続けている [53, 54, 64, 66, 76]．一方で，計算リソースが制限された環境へ向けた言語機能制限あるいは機能の不足から，記述性が損なわれたり不安定な振る舞いを示したりすることがあった．例えば，Emfrp では一般的な関数型言語で頻繁に使用される再帰データ型や再帰関数の使用が許されていない．これは実行途中の無限ループを避け，システム全体の応答性を保証するためである．しかし，この制限により一般に知られたプログラミング技法を活用できず，不自然なプログラム記述を強いられることがあった．また，Emfrp では言語ランタイムを簡素にする目的から，ある種のポーリングによって時変値更新処理を行うことでシステムの応答性を担保している．ところが，その仕組みのために時変値更新処理の途中に CPU リソースが占有され応答性が低下する問題や必要以上の電力を消費してしまう問題があった．

前述した組込みシステム向け FRP 言語のいくつかは Emfrp に類似した実行モデルを持つため，これらの制限や問題は共通している．また，これらの制限が緩和されている言語についてはメモリに関する安全性の保証がないなど，デバッグのしづらい組込み環境での使用に難のある傾向があった．そこで，本研究では計算リソースの適切な管理機構によって上記の問題を解決できるのではないかと予想を立て，組込み環境下で特に重要な 3 つの計算リソース (メモリリソース，CPU リソース，電力リソース) に注目した．これらのそれぞれのリソースをより詳細に扱い管理する機構を持った拡張言語を 3 つ提案し，上記の問題への解決策を示した．本研究の貢献は以下である．

- 小規模な組込み環境を対象にした FRP 言語 Emfrp において，メモリリソース，CPU リソース，電力リソースのそれぞれを適切に管理する機構が不足していた場合に，どのような問題点が起きるのかを分析し具体例として提示した．
- それぞれの問題点に対し，適切なリソース管理機構を導入して拡張した新たな FRP 言語 $\text{Emfrp}^{\text{BCT}}$，$\text{Emfrp}^{\text{HT}}$，EvEmfrp/S を設計した．
- 拡張言語による記述では，先述の問題点が緩和あるいは解決することを具体例を通して示した．

- 特に，メモリリソースについて注目した拡張言語 $\text{Emfrp}^{\text{BCT}}$ については，言語の形式化および処理系の評価をおこなった．

1.3 論文の構成

以下に本論文の構成を述べる．

- 第 2 章では，本研究が対象とする小規模組込み環境とドメイン特化言語による開発支援について説明したのち，本研究の基礎となる Emfrp 言語について解説する．
- 第 3 章では，Emfrp 言語に対し適切なメモリリソースの検査や管理機構を導入することで，再帰データ型や再帰関数に関する制限が緩和できることを示す．より具体的には「構築できる大きさに制限をつけた再帰データ型」をもつ Emfrp の拡張言語 $\text{Emfrp}^{\text{BCT}}$ を提案する．
- 第 4 章では，Emfrp 言語の実行モデルでは，CPU リソースが占有されることでシステム全体の応答性に影響が出ることをロボット競技の具体例を通して分析する．その問題への解決手法として非同期タスク実行を導入した拡張言語 Emfrp^{HT} を提案する．
- 第 5 章では，周期的な間欠実行によって消費電力の削減を図った Emfrp の後続言語 EvEmfrp に対し，その実行周期の柔軟性が不足していることで消費電力の削減が十分に行われないことを指摘する．この問題への解決策として，非同期イベントを起点にして実行周期を動的に切り替える機構を持つ拡張言語 EvEmfrp/S を提案し，具体例を通して電力削減の機会が増加していることを示す．

最後に第 6 章にて，以上の 3 つのリソースアウェアな機構の統合について考察したのち，全体をまとめる．

第 2 章

背景と関連研究

2.1 小規模組込みシステムの開発

本研究でのプログラムの実行対象となる小規模組込み環境について説明する．本研究が扱う性能の範囲をもつマイクロコントローラ (MCU) を表 2.1 に例示した．マイクロコントローラとは，計算の主体となる CPU の他にメモリやアナログデジタル変換器などの周辺機器 (ペリフェラル) を備えた IC チップを指す．Arduino [3] はマイコンボードの形でホビー向けに提供されることが多く，Flash も RAM もリソースが限られたデバイス環境である．STM32 マイコン [68] は車載やホビー向けに提供されており，バージョンによって異なるが Flash は数十から数百 KB ほど，RAM は多くても数百 KB である．Raspberry Pi Pico [51] や ESP32 [69] では，Flash や RAM 容量が比較的大きいため，メモリ配置などに気を配らなくてもある程度自由にプログラミングできると予想される．

一般にマイクロコントローラをプログラミングする際には，C/C++ によってオペレーティングシステム (OS) の搭載されていない環境をベアメタルプログラミングするか，FreeRTOS [61] や Zephyr [17] などのリアルタイム OS を搭載しタスクベースのプログラミングをするかのどちらかである．どちらの場合でも，基本的に実行中のメモリ破壊による不具合やリソース競合を発見するのは難しく，デバッグには時間を要する．

プログラムを実行する上で特に大切なのは RAM の大きさである．DSL の実行にはユーザープログラムの他に DSL の言語ランタイムが動作する．オリジナルの Emfrp や組込み向け FRP 言語 Juniper [25] は Arduino 上でも動作可能なほど，言語ランタイムおよびそのフットプリントが小さくなるように設計されている．

これらの MCU を CPS におけるエッジデバイスとして使用する場合には，センサーやアクチュエータの他に WiFi や ZigBee などのネットワークに接続するための外部デバイスが付随する．

表 2.1　対象とするマイクロコントローラの性能

MCU	Clock(max) [MHz]	Flash [KB]	RAM [KB]	CPU
Arduino UNO R3	16	32	2	AVR ATmega308P 8bit
STM32L101RBT6	32	128	20	ARM Cortex M0+ 32bit
STM32F303K8T6	72	64	16	ARM Cortex M4 32bit
STM32F401RET6	84	512	96	ARM Cortex M4 32bit
Raspberry Pi Pico	133	2048	264	RP2040 (ARM Cortex M0+ x 2) 32bit
ESP32	240	448	520	Xtensa dual-core LX6 32bit

2.2 ドメイン特化言語による開発

組込みシステム向け FRP 言語を使用してエッジデバイスを記述することで，ドメイン特化言語を使った開発での利点が享受できる．ドメイン特化言語を利用した開発には以下の利点がある．

- 対象のドメインに固有の書き方でプログラムが記述できるため，可読性や保守性に富んだプログラムコードによる開発が行える．
- その言語で書くことで，「良い性質」が保証される場合がある．
- 一つのソースコードから複数の動作対象 (プラットフォーム) に対して，コード生成をすることができる．
- ユーザーが記述したプログラムにおいて，言語ごとに特有のチェックをすることができる．

例えば，YACC や Menhir などのパーサージェネレータ上の DSL では，構文規則を BNF に近い形で記述することができ，可読性に富んだプログラムコードによる開発が行える．また，パーサーが生成される前にパーサージェネレータによって，文法が曖昧かどうかの検査が行われ予期せぬ間違いを見つけられる可能性がある．組込み開発分野での他の例としては MATLAB/Simulink が挙げられる．これはテキストベースのドメイン特化言語ではないものの，モデリングおよびシミュレーション用のコードから実際にマイコン上で動作する C コードを生成することができる．

典型的なリアクティブシステムであるエッジデバイスを組込みシステム向け FRP 言語で記述することは妥当な選択である．

2.3 関数リアクティブプログラミング

関数リアクティブプログラミング (Functional Reactive Programming, FRP) は，組込みシステムや GUI などの**リアクティブシステム**を簡潔に記述するためのプログラミングパラダイムである．リアクティブシステムとは，非同期的な入力に対して処理を行い，内部状態を変化させながら外部への出力をし続けるシステムである．リアクティブシステムと見なされるものは多くあり，家電，テレビゲーム，自動車なども含まれる．これらのリアクティブシステムは手続き型プログラミングの文脈では，一般にポーリングやコールバック (割り込み) を用いてプログラムされる．ところが，これらのプログラミング技法はプログラムの制御フローや，値や処理の依存関係の把握を困難にすることが知られており，コード理解の妨げや保守性に乏しくなるなどの問題点があった．

FRP では**時変値 (time-varying value)** と呼ばれる時間と共に自動的に変化するオブジェクトを宣言的に組み合わせることで，リアクティブシステムを簡潔に記述する．FRP のユーザーはシステムの入力から出力までのデータフローが明示されることで，簡潔かつ保守性の高いコードの記述が可能である．

時変値にもとづく FRP の考え方が初めて導入されたのは Elliot らによる Haskell 用の対話的アニメーションライブラリ Fran [14] である．Fran による初期の FRP の実装では，時変値は時刻から値への関数として扱われ，任意の時刻に対する時変値を取得することが可能であった．そのため，過去の時変値の値を保持し続けることでメモリ消費量が時間と共に増加してしまう**空間漏れ (space-leak)** や，時変値の更新の際に使用される履歴の長さが増えていくことで計算時間が増加してしまう**時間漏れ (time-leak)** などの問題点があった．これらの問題を解決するために FRP の計算モデルが複数提案され [74, 75] た．Yampa [31] では時変値上の関数を第一級として扱い，Arrow [32] と呼ばれる構造上に FRP の計算モデルを構築し，時変値そのものを取り扱うことを禁止することで，この問題を解決した [42].

```
module RobotPos    # module name
in  vl : Float,    # left wheel speed (m/sec)
    vr : Float,    # right wheel speed (m/sec)
    theta : Float, # angle from x-axis (rad)
    t(0) : Int     # elapsed time (msec)
out x : Float      # x-position of the robot (m)
use Std            # library name

# a small amount of elapsed time (sec)
node dt = (t - t@last) / 1000.0

# x-position of the robot (m)
node init[0.0] x =
  x@last + (vr+vl) * cos(theta) * dt/2
```

図 2.1　Emfrp による差動二輪ロボットの位置計算プログラム

現在，FRP はライブラリ [2,31,65] や言語 [12] として実装され，GUI [12] やロボティクス [31,50] などの分野で使用されている．

2.4　Emfrp

Emfrp は小規模組込みシステムで動作するリアクティブシステムを記述するための純粋な静的型付き FRP 言語である．RAM や Flash(ROM) が数 KB 程度に限られ，十分な CPU 性能もないマイクロコントローラのような環境や，OS などのメモリ管理プロセスが存在しないベアメタル環境下での実行を想定している．

2.4.1　コンパイルフロー

Emfrp プログラムはリアクティブな処理を連続して行うためのループが記述された C 言語のソースファイルと入出力のためのテンプレート (関数定義) が記述された C 言語のソースファイルにコンパイルされる．ユーザーは外部環境と Emfrp 上のモジュールへの入出力を行うテンプレート部分を別途 C 言語で記述して，ループを実行することで，Emfrp モジュール全体を実行することができる．入出力部分は環境に依存しやすいことが想定されるが，Emfrp のプログラムは比較的環境依存の少ない C 言語のコードへとコンパイルされるため，適切に入出力部分を記述することで，様々なプラットフォーム上で動作させることが可能である．

2.4.2　プログラム例

Emfrp で記述した差動二輪ロボットの位置を計算するプログラムを図 2.1 に示す．これは文献 [31] の例を Emfrp で記述したものである．このプログラムは入力として左右の車輪の速度 v_l, v_r，車体と x 軸との角度 θ，経過時間 t を入力として受け取り，車体の x 座標 x を出力する．入力と出力の関係は以下の式によって表される．

$$x(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u))\cos(\theta(u))\,du \tag{2.1}$$

Emfrp プログラムはモジュールという単位で記述される．1 行目はモジュールの名前を記述している．Emfrp では時変値は**ノード**と呼ばれる．ノードは入力ノード，出力ノード，および中間ノードに分類される．入力ノードは外部の環境からの値を取り込み時変値とし，出力ノードは計算された時変値を

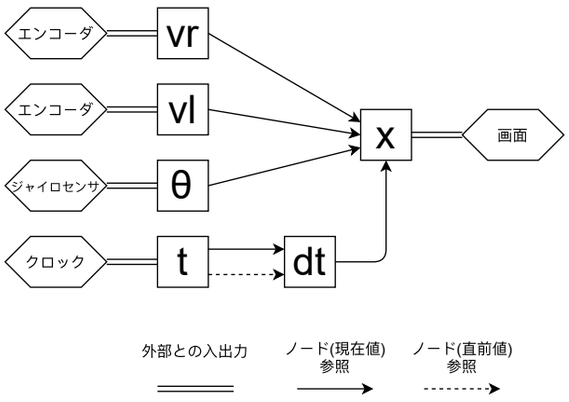

図 2.2　RobotPosモジュールの依存グラフ

外部の環境へ出力する．

図 2.1 の 2–5 行目は入力ノード (vl，vr，theta，t)，6 行目は出力ノード (x) の宣言であり，関係式 2.1 中の (v_l, v_r, θ, t, x) と対応する．入力ノードの値は外部の環境から与えられる．この例では，vl，vrには車体に搭載されたロータリーエンコーダの値 (車輪の速度) が，thetaにはジャイロセンサーの値 (車体の角速度) が，tには CPU などで計測された経過時間が入力されることが想定される．中間ノードおよび出力ノードの値は予約語 nodeを用いて定義される．10 行目，13 行目ではそれぞれ中間ノード dtと出力ノード xを定義している．ノードの定義は node n = e または node init[c] n = e の形で定義される．n はノード名，e はノードの関係を表す更新式である．node init[c]はノードの初期値を指定する記述である．ノード名に@lastを付けると，そのノードの**直前値**を取得することができる．例えば 10 行目の t - t@lastでは前回のノード更新からの時間 (時刻の差分) が計算される．13 行目では x@lastに対し，微小変化量を加算し続けることで時間 tによる積分を計算する．@lastで参照されるノードには初期値の指定が必須である．入力ノード tの初期値は 5 行目で 0 に指定され，出力ノード xの初期値は 13 行目で 0.0 と指定されている．

2.4.3　実行モデル

ノード n の定義中 (更新式中) にノード n' が使用されている場合，「n は n' に依存する」という．Emfrp プログラムではノード間の依存関係が循環しないことが要求される．つまり，ノードを頂点，ノードの依存関係を辺として構成したグラフが有向非巡回グラフになることが要求される．このグラフを依存グラフと呼ぶ．このとき，@lastによる使用は依存関係の辺が張られないことに注意されたい．例えば，図 2.1 の依存グラフは図 2.2 になる．この図中では依存関係の辺は実線で，@lastの使用を表す関係は点線で表示されている．

Emfrp コンパイラは依存グラフをトポロジカルソートすることで，ノードの更新順を静的にスケジューリングする．図 2.2 からは例えば (t，dt，vl，vr，theta，x) がノード更新の順序として得られる．外部からの入力を受け取り，得られた更新順にひとつずつノード更新を行い，外部へ値を出力，最後にメモリ管理を行う．この一連のノード更新とメモリ管理をイテレーションと呼ぶ．Emfrp によって記述されたプログラムは，イテレーションを繰り返すことでリアクティブな挙動を行う．このように依存関係の根元からノードの更新処理を行う FRP の実行モデルは push 型 [4] と呼ばれる．イテレーション中に更新の必要がない時変値を更新してしまう可能性があるが，スケジューリングやランタイムの実装が簡素であるため，小規模環境を想定している Emfrp において採用されている．

2.4.4 メモリ管理

Emfrp でタプルや直和型を持つオブジェクトはその型ごとの配列として静的に確保された領域 (ヒープ領域) に格納される．プログラム中の関数呼び出しでは，これらのヒープ領域への参照渡しが行われる．また，`Int`型などのプリミティブな型のオブジェクトは C 言語での対応した基本型にコンパイルされ，ヒープ領域を使用せずに計算が行われる．

プログラム中で必要のなくなったオブジェクトを適宜解放するために，ノード更新やイテレーションの途中で，ある種のマークスイープ GC が行われる．それぞれのノード更新の直後には更新されたノードが確保されたヒープ領域にオブジェクトの生存時間を延長するためのマークが行われる．ヒープ上のオブジェクトはアロケーションの際に設定される生存時間によって領域の使用，不使用が判断される．そのため，ノード更新の際に一時的に使用されるオブジェクトの明示的な解放は行われない．イテレーションの最後には直前値が確保されているヒープ上の領域を解放し，現在値ノードを直前値ノードとして切り替える処理が行われる．

Emfrp では，それぞれのノードの現在値と直前値が使用するメモリ量，ノードの更新中に使用する最大メモリ量，言語ランタイムが使用するメモリ量の合計が実行時に必要なメモリ量である．Emfrp では実行時に必要となるメモリ量を静的に決定できるため，オブジェクトのためのヒープ領域やローカル変数のためのスタック領域の大きさも静的に定めることが可能である．したがって，ヒープ領域は実行時に伸長されることはなく，静的に確保された一定量のメモリ領域内でプログラムの実行を行うことができる．

2.4.5 機能制限による性質の保証

時変値の過去の値の参照を直前値のみに限定する，時変値を第一級として扱わず必ず名前で参照する，高階時変値を扱わない，関数内部での時変値の参照を禁止するといった制限を行うことで，過去の時変値の値を保持し続けることでメモリ消費量が時間と共に増加してしまう**空間漏れ (space leak)** や時変値の更新の際に使用される履歴の長さが増えていくことで計算時間が増加してしまう**時間漏れ (time leak)** といった FRP 特有の問題が起こらない設計となっている．

第 3 章

メモリリソース：$\mathrm{Emfrp}^{\mathrm{BCT}}$

3.1 まえがき

本章の主な貢献は**使用するメモリ量の情報を含んだ再帰データ型を取り入れた FRP 言語 $\mathbf{Emfrp}^{\mathbf{BCT}}$ を設計し，その言語のメモリ使用量決定アルゴリズムを提案した**ことである．より具体的には以下が挙げられる．

- $\mathrm{Emfrp}^{\mathrm{BCT}}$ の構文，操作的意味論，型システムについて形式的定義を与えたのち，式の評価に必要なメモリ量を決定するアルゴリズムを提示した．
- $\mathrm{Emfrp}^{\mathrm{BCT}}$ の定義を用いて，「適切に型付けされサイズの妥当性の検査に通ったノードについて，使用メモリ量決定アルゴリズムによって得られたリソース量で十分にノード更新処理が行える」ことを証明した．
- $\mathrm{Emfrp}^{\mathrm{BCT}}$ から C 言語へのトランスパイラを作成し評価実験として，型検査や使用メモリ量の決定にかかる時間の測定，再帰データ型を使用することによる実行時の時間的，空間的オーバーヘッドの測定を行った．

$\mathrm{Emfrp}^{\mathrm{BCT}}$ は，前述した Emfrp の持つ性質を維持しながらも，再帰的定義に関する制限を緩和するために**「構築できる構造の最大サイズ」を伴った再帰データ型**で Emfrp を拡張した言語である．$\mathrm{Emfrp}^{\mathrm{BCT}}$ では，再帰データ型や関数呼び出しの際の引数のサイズが減少することを保証した再帰関数 (原始帰納的関数) を利用することができる．ここで導入されたデータ型とそれに付随する言語機構は，先行研究である Sized Types [34] を Emfrp のプログラミングスタイル (直前値を積極的に活用するスタイル) に合わせて拡張したものである．提案手法と Sized Types の詳しい比較は 3.6.1 節にて行う．

$\mathrm{Emfrp}^{\mathrm{BCT}}$ では，型検査の際にサイズに関する検査が同時に行われるため，メモリを無制限に使用するプログラムがコンパイル，実行されることはない．また，ノード更新の停止性も保証されるため，従来の Emfrp と同様にメモリリソースが不足することなく，リアクティブな動作をし続けることが静的に保証できる．さらに，従来の Emfrp ではデータをタプルによって管理せざるを得なかったプログラムが，$\mathrm{Emfrp}^{\mathrm{BCT}}$ によってリストやヒープ木を用いて簡潔かつ保守性に富んだ記述で実現できることを例題を通して示した (第 3.3.4 節)．

再帰的定義に関する制限が緩和されたことで，上記のように関数プログラミングの文脈で使われている技法を自然な記述で，リアクティブプログラミング言語に取り入れることができる．自然な記述は可読性の高いプログラムコードの生産に貢献し，それは安定なシステム構築の生産性向上へと寄与する．

本章の主な内容は文献 [77] にて発表済みである．

```
1  (* 「式」の型定義 *)
2  type expr = Int of int
3            | Add of expr * expr
4            | Mul of expr * expr
5
6  (* 「式」の計算 *)
7  let rec calc (e : expr) : int =
8    match e with
9    | Int i -> i
10   | Add (e1, e2) -> calc e1 + calc e2
11   | Mul (e1, e2) -> calc e1 * calc e2
12
13 let nine = calc (Mul (Add (1, 2), 3))
```

図 3.1　OCaml による再帰データ型の定義と使用の例

3.2　動機：再帰データ型を使用したプログラミング

3.2.1　再帰データ型

再帰データ型[*1]は同じ構造を再帰的に含むデータ構造を表す型である．例えばリスト構造や木構造を表すために再帰データ型が用いられる．組を表す直積型とタグによって内容を判別できる union(disjoint union) を表す直和型を組み合わせたデータ型をヴァリアント型と呼ぶ．ヴァリアント型では静的に決められた大きさのデータ構造しか表すことができない．そこで再帰データ型では，データ構造の再帰性に注目しヴァリアント型の宣言時に自己言及を許すことで，実行時に要素数の変わりうるデータ構造に (静的に決定する) 型を付ける．再帰データ型およびそれを分解するための機構 (再帰関数) や構文 (パターンマッチ) は，多くの関数型言語で基礎的な機能として扱われる．

関数型言語 OCaml による再帰データ型の利用例を図 3.1 に記載する．1 行目で定義されている`expr`型は再帰データ型である．`Int`や`Add`は直和型のタグ，`*`は型の直積を表す記法である．2 行目と 3 行目では`expr`型の定義中に`expr`が再帰的に現れていること (自己言及) がわかる．このような型定義によって，加算と乗算のある数式の有限長の AST をその長さにかかわらず扱うことができる．5 行目ではデータ型をパターンマッチによって分解し，再帰的に値を計算する関数`calc`を定義している．再帰データ型を使用する際には，データの構築や分解を行うために再帰関数やパターンマッチ機構と組み合わせて使用されることがほとんどである．

3.2.2　Emfrp の制限

2.4 節で説明したように Emfrp は記述したプログラムの実行に際し**ノード更新処理の停止性**と**実行時に必要とされるメモリ量を静的に決定できる**性質を保証する．これにより，Emfrp プログラムは計算資源の乏しい環境下でも安全にリアクティブな振舞いを続けることが静的に保証されている．これらの性質を保つために Emfrp ではデータ型や関数を定義する際に再帰的な定義を禁止している．Emfrp 上で使用できるデータ型は整数値や真偽値などを表すプリミティブな型，タプルを表す直積型，タグによって内容を判別できる直和型に限られ，再帰関数は定義できない．データ型の再帰的定義を禁止することでプログラムの実行に必要なメモリ量の静的な算出が容易になり，再帰関数を禁止することでノード更新の停止性を保証することが容易になる．そのため，**Emfrp 上では実行時に要素数が変更されうるデータ構造を自然な表現で扱うこと**が問題となる．

[*1] あるいは代数的データ型 (Algebraic Data Types，ADT) と呼ぶ．

```
module DupCheck
in  reset(False) : Bool, # reset signal
    v(0)         : Int   # input value
out detect : Bool
use Std

type OptI = NoneI | SomeI(Int)

func check(x : Int, a : OptI): Bool =
  a of NoneI -> False,
       SomeI(y) -> x == y

node init [(NoneI, NoneI, NoneI, NoneI)]
  history: (OptI,OptI,OptI,OptI) =
    if reset then (NoneI, NoneI, NoneI, SomeI(v))
    else
      history@last of (a1, a2, a3, a4) -> (a2, a3, a4, SomeI(v))

node init [False] detect: Bool =
  history@last of (a1, a2, a3, a4) ->
    check(v,a1) || check(v,a2) || check(v,a3) || check(v,a4)
```

図 3.2　Emfrp による DupCheckモジュールの実装

前述した直積型を使用することで，要素数が不変なデータ構造を扱うことは容易である．一方で，リスト構造や木構造のような実行時に要素数が変更されるデータ構造を，Emfrp 上で使用できる型の組み合わせで定義するためには，非常に冗長な記述が要求されるか，あるいはそのようなデータ構造を定義することができない．一般に FRP と同様に宣言的な記述によってプログラミングを行う関数型言語では，そのようなデータ構造は再帰データ型によって定義され，頻繁に使用されている．

3.2.3　問題となる例

小規模な組込みシステムにおいてもリスト構造や木構造が有用であるようなプログラム例を 2 つ提示し，既存の Emfrp では冗長な記述が要求されることを説明する．

DupCheckモジュール

直近に入力されたデータが過去に入力されているかを判定するモジュール，DupCheckモジュールを例にあげる．このモジュールは今まで過去 4 イテレーション分に入力された整数データを履歴として保持している．入力された整数データが履歴に存在するかどうかを検査して，その結果を真偽値として出力する．Emfrp を用いて記述した例を図 3.2 に示す．

このモジュールは入力時変値として，リセットシグナル resetと入力整数値 vを持つ．モジュールの先頭にて Int型に対するオプション型を定義している．入力履歴は historyノードによってオプション型のタプルとして表される．最初の 4 イテレーションはデータが存在しない部分があるため，オプション型を用いている．resetシグナルが真になっている場合には現在値のみを履歴として設定する．resetシグナルが偽の時には通常の履歴更新処理を行う．履歴列の先頭 (最古の履歴) を削除して，末尾 (最新の履歴) に入力された値を追加する．履歴に入力値が存在するかを表す真偽値は detectノードによって表される．detectノードの更新式では履歴をタプルのパターンマッチによって分解し，check関数を用いて存在判定を行っている．この時，過去の履歴を参照するために history@lastを分解することに注意されたい．

履歴管理にタプルを使用しているため，履歴長に変更があった場合に変更しなければならない箇所

```
module Top10Sum
in  reset(False) : Bool,
    v(0) : Int           # input value
out sum : Int            # sum
use Std

type OptI = NoneI | SomeI(Int)
type TI10 = T10(OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI)

func gt_opt(x : Int, a : OptI): Bool =
  a of NoneI -> True, SomeI(y) -> x > y

func get_opt(x : OptI): Int =
  x of NoneI -> 0, SomeI(y) -> y

node init [T10(NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI)]
  h: TI10 = if reset then T10(NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI,
    NoneI, NoneI)
    else
      h@last of T10(a0, a1, a2, a3, a4, a5, a6, a7 ,a8, a9) ->
           if gt_opt(v, a0) then T10(SomeI(v), a0, a1, a2, a3, a4, a5, a6, a7, a8)
      else if gt_opt(v, a1) then T10(a0, SomeI(v), a1, a2, a3, a4, a5, a6, a7, a8)
      else if gt_opt(v, a2) then T10(a0, a1, SomeI(v), a2, a3, a4, a5, a6, a7, a8)
      else if gt_opt(v, a3) then T10(a0, a1, a2, SomeI(v), a3, a4, a5, a6, a7, a8)
      else if gt_opt(v, a4) then T10(a0, a1, a2, a3, SomeI(v), a4, a5, a6, a7, a8)
      else if gt_opt(v, a5) then T10(a0, a1, a2, a3, a4, SomeI(v), a5, a6, a7, a8)
      else if gt_opt(v, a6) then T10(a0, a1, a2, a3, a4, a5, SomeI(v), a6, a7, a8)
      else if gt_opt(v, a7) then T10(a0, a1, a2, a3, a4, a5, a6, SomeI(v), a7, a8)
      else if gt_opt(v, a8) then T10(a0, a1, a2, a3, a4, a5, a6, a7, SomeI(v), a8)
      else if gt_opt(v, a9) then T10(a0, a1, a2, a3, a4, a5, a6, a7, a8, SomeI(v))
      else h@last

node init [0] sum: Int = h of T10(a0, a1, a2, a3, a4, a5, a6, a7 ,a8, a9) ->
  get_opt(a0) + get_opt(a1) + get_opt(a2) + get_opt(a3) + get_opt(a4)
  + get_opt(a5) + get_opt(a6) + get_opt(a7) + get_opt(a8) + get_opt(a9)
```

図 3.3　Emfrp による Top10Sumモジュールの実装

が非常に多くなってしまう．具体的には historyノードの型や初期値，更新式中のパターンマッチ，detectの更新式中のパターンマッチの部分である．コードを変更する際には，パターン中の変数名を適切に書き換える必要がある．ところが，それぞれの変数は全て同じ型であるため，書き間違えがあったとしてもその間違えを検出することができない．そのため，既存の Emfrp によるタプルを用いた方法では保守性に乏しいだけでなく，バグを埋め込みやすい冗長な記述となってしまう．

Top10Sumモジュール

Top10Sumモジュールは今までの上位 10 個の入力値の和を出力するモジュールである．この例でも，DupCheckモジュールと同様に入力履歴を用いて結果を計算する．タプルとオプション型を用いて記述した例が図 3.3 である．

ノード hは降順にソートされた上位 10 個の入力値を保持するノードである．オプション型のタプルによって表されている．ノード hの更新の際には，挿入ソートの要領で，hの要素と入力値 vの大小によって vを挿入すべきかどうか決定する．出力ノード sumは h中の要素の合計を計算する．このモジュールも DupCheck モジュールと同様に非常に冗長な記述となっている．

Emfrp で記述した例では入力値 vの挿入箇所を探す処理の時間計算量は O(履歴の要素数) となっているが，ある集合の中から最小値を探す際には木構造 (ヒープ木) を用いることで処理の時間計算量を抑

えることができる．そのため処理すべき履歴が多くなった際には木構造による管理が有効である．しかし，既存の Emfrp によるタプルと直和型によるデータ表現では木構造の表現を行うことは困難である．以上の例から，実行時に要素数が変更されるデータ構造が扱いにくいと言う欠点は，ソースコードの記述が冗長になるだけでなく，ノード更新計算のパフォーマンスに影響を及ぼす可能性が示唆される．

3.3 $\text{Emfrp}^{\text{BCT}}$

3.3.1 解決方針

Emfrp に対し再帰データ型を導入することで，前述した問題点を解決する．しかし，関数型言語で一般に用いられているような再帰データ型や再帰関数定義を単純に導入することはできない．制限のない再帰データ型や再帰関数を単純に導入した場合，以下のようなノードや関数を定義することができる．

```
1 # Node with recursive data types (lists for nats)
2 node init[Nil] r : IntList = Cons(x, r@last)
3 # Infinite recursion
4 func infloop(x : Int) = 1 + infloop(x)
5 node init[0] i : Int = infloop(0)
```

ノード `r`は初期値としてリストの末尾 `Nil`を持ち，ノード更新のたびに入力値 `v`を自身に連結し続ける．そのため，時間と共にメモリの使用量が増大し，メモリを無制限に使用してしまう．ノード `i`のように，計算が停止しない関数 `infloop`を更新式中で呼び出すことで，更新計算が停止しなくなってしまう．これらの例では，Emfrp の持つ静的に使用メモリ量を決定できるという性質やノード更新処理の停止性の保証ができなくなっている．したがって，再帰的定義の単純導入は Emfrp が保証する性質を壊してしまう．そこで，構築できる構造の大きさや，再帰呼び出しの回数についての制限をもった再帰的定義と，この制限に違反する記述を静的に検出する仕組みを Emfrp に対して導入する．構造の大きさに構築制限をかけることで，前述した例のように無制限に伸び続けるリストの記述を検出することができる．さらに再帰呼び出しの回数が制限されることで再帰関数の計算の停止性が保証され，実行時に必要なコールスタックの量やその他のメモリ使用量も静的に計算することができるようになる．

3.3.2 Bounded-Construction-Types

本研究では，構築できる構造の大きさ (サイズ) を型として表現する．以降ではデータ構造の大きさを，構造中に含まれるそのデータ構造の値コンストラクタの数とし，これをサイズと呼ぶ．例えば以下の OCaml のプログラム中の `ilist`型の変数 `l`のサイズは 4 であり，変数 `n`のサイズは 1 である．同様に `otree`型の変数 `t`のサイズは 5 である．

```
1 type ilist = Nil | Cons of int * ilist
2 let l = Cons (1, Cons (2, Cons (3, Nil)))
3 let n = Nil
4
5 type otree = L | N of otree * int option * otree
6 let t = N (N (L, Some 2, L), Some 1, L)
```

サイズは正の整数をとり，対象とする型のコンストラクタの数のみを数えることに注意されたい．

提案手法において，サイズについての情報が付与された再帰データ型を **Bounded-Construction-Types (BCT)** と呼ぶ．BCT が持つサイズについての情報とは，「**構築できる構造の最大サイズ**」である．このサイズについての情報を**サイズパラメータ**と呼ぶ．例えばサイズ 4 までの構築を許す整数リスト型は $\texttt{ilist}^4$ のように記述される．上記例の変数 `l`や `n`は $\texttt{ilist}^4$ 型のインスタンスである．変数 `l`はサイズ 4 なので，$\texttt{ilist}^2$ のインスタンスではないが，変数 `n`はサイズ 1 なので，$\texttt{ilist}^2$ のインスタ

ンスである．同様に変数 tはサイズ 5 なので otree^{8} のインスタンスであるが，otree^{4} のインスタンスではない．

このように実行時に構造の大きさが変わるデータ構造に対し構築数の制限を設け，それを型として表現することで，サイズが無制限に増えるオブジェクトを型検査によって静的に検出できる．さらに，再帰関数について，その内部での再帰呼び出し時に引数が構造的に減少しているかどうかサイズパラメータを用いて型検査時に静的に検査する．再帰関数呼び出しの際に入力の構造が減少していることが保証されると，その構造のサイズ分しか再帰関数が呼び出されないことがわかる．これにより，再帰関数呼び出しに対して整礎性が要求され，呼び出しの停止性を静的に保証することができる．

BCT の型検査の際には通常の型検査で行われる値の種類の整合性の検査に加え，構造の取りうる大きさ (サイズパラメータ) の整合を検査するための制約抽出が行われる．その制約が妥当であれば，記述されたプログラムがサイズや再帰呼び出しの制限に関して違反していないことが保証される．

構築できるサイズの**最大数**に注目している理由は，BCT 型ノードがイテレーションごとに保持する要素数が変化することを許容するためである．もしこれを許容せずサイズが一致する型を採用したとすると，イテレーションごとにサイズを合わせるためのダミーデータの挿入が要請される．これを避けるために上記の BCT を採用している．

3.3.3 Emfrp$^{\mathrm{BCT}}$ の言語機構

既存の Emfrp のサブセットに対し BCT を組み込んだ言語，Emfrp$^{\mathrm{BCT}}$ を提案する．Emfrp$^{\mathrm{BCT}}$ は構文や実行時のデータ表現に差異があるものの，基本的な実行モデルは同一である．Emfrp では関数に多相型を持つことがあるが，本研究ではサイズパラメータに注目するために扱わないものとする．Emfrp$^{\mathrm{BCT}}$ では高階関数が利用できないため，多相関数からは型変数で表されたデータに対するサイズの制約が生成されない．そのため，仮に Emfrp と同様の多相型を導入するとしても多相関数については通常の多相型理論と同程度の検査で十分であり，本質的には難しくない。

Emfrp$^{\mathrm{BCT}}$ では，型検査に成功すると，プログラム中の値のサイズ情報に整合性が取れていることが保証される．その情報を用いて構文木を網羅的に走査することで，プログラムが実行時に必要とするメモリ量を静的に計算することができる．したがって，Emfrp$^{\mathrm{BCT}}$ は再帰データ型 (BCT) を使用可能でありながら，時変値更新処理の停止性と実行時のメモリ使用量を静的に決定できる性質を保存している．以降の節では，Emfrp$^{\mathrm{BCT}}$ を各機能ごとに説明する．

モジュール定義

モジュール定義は Emfrp と同様にデータ型，関数，入力ノード，出力ノード，中間ノードの定義を行う．Emfrp では型推論があったため幾らかの型注釈が省略可能であったが，Emfrp$^{\mathrm{BCT}}$ では全ての型を注釈する必要がある．

型定義

Emfrp$^{\mathrm{BCT}}$ では `Int`型のような基本型に加えて，再帰データ型として BCT を定義することができる．Emfrp で使用できたタプル型，直和型，およびそれらを組み合わせたヴァリアント型は再帰しない BCT として同等の機能が模倣できるため直接言語への導入は行われていない．BCT の型定義は以下のような記述で行うことができる．

$$\texttt{type}\ \rho\ \texttt{=}\ \chi_1(T^{\rho}\ ,\ldots,\ T^{\rho})\ \texttt{|}\ \ldots\ \texttt{|}\ \chi_n(T^{\rho}\ ,\ldots,\ T^{\rho})$$

ρ は型名，χ はコンストラクタ名を表す．型名やコンストラクタ名はプログラム中でユニークである．T^{ρ} はコンストラクタのパラメータとして持つことのできる型であり，基本型か ρ か他のサイズを伴っ

た BCT を指定できる．ρ はパラメータのどの位置が再帰している型なのかを表すためのプレースホルダとして扱われる．本稿では，**BCT 定義の際の相互再帰は扱わない**ことに注意されたい．すなわち，再帰的な型の定義の際には，必ず自身の型名がいずれかのコンストラクタのパラメータに現れることになる．

例えば BCT で整数を要素に持つ二分木は以下のように定義することができる．

```
type TreeI = LeafI | NodeI(TreeI, Int, TreeI)
```

BCT を定義する際にコンストラクタのパラメータに他の BCT が現れることも許容している．その際には必ず定数のサイズパラメータが設定されていることが要求される．例として，サイズ 10 までのリスト型を要素として持つ二分木の型定義を以下に示す．

```
type TreeL = LeafL | NodeL(TreeL, ListI[10], TreeL)
```

BCT の名前は自身の型定義に登場する際にはサイズパラメータを指定しない．しかし，他の型定義中や関数の引数で型名として登場する場合には名前に `[`ψ`]`を追加してサイズパラメータを指定する．ψ はサイズパラメータを表すメタ変数である．サイズパラメータは定数，サイズパラメータ同士の和差，サイズ変数を表す．

BCT のコンストラクタを値に対して適用すると，得られる値は適用された値のサイズの合計よりもサイズが 1 だけ大きな型をもつ．つまり，上記の例の `LeafI`という値は `TreeI[1]`型を持ち，`NodeI(NodeI(LeafI, 2, LeafI), 1, LeafI)`や `NodeI(LeafI, 3, NodeI(LeafI, 2, LeafI))`は `TreeI[5]`型を持つ．

関数定義

Emfrp$^{\mathrm{BCT}}$ では，Emfrp で許されていた関数定義に加えて，再帰的な関数定義を行うことができる．ただし，再帰的定義は直接的に行われるものとし，**相互再帰的な定義は扱わない**．また，再帰関数の引数は再帰呼び出しの際にサイズが小さくなるという制約がある．

以下の記述により，再帰関数を定義する．

```
func f( x_1:π_1 ,..., x_n:π_n ):τ where {A_1,...,A_m} [δ_1,...,δ_l] = e
```

f は関数名，x は引数名である．π は関数引数に指定できる型であり，基本型かサイズパラメータとしてサイズ変数を伴った BCT を取る．ここで導入されたサイズ変数は関数本体の式の `adj`式 (後述) で使用することができる．τ は結果の型，$\mathcal{A}$ はサイズ変数に関する論理式であり，関数呼び出しに対し静的に検査されるべき事前条件を表す．δ はサイズ変数であり，引数に登場するサイズ変数である．これは再帰呼び出しにおけるサイズ減少の尺度関数を表し，`[`$\delta_1,\ldots,\delta_l$`]`内に指定されたサイズ変数の和が再帰呼び出しの際に小さくなることが関数定義時に検査される．e は関数本体の式である．関数定義中にはノード参照やノードの直前値の参照は許されていない．

関数定義は「定義時に指定した事前条件を前件とし，本体の式から得られたサイズに関する制約を後件」とした含意の真偽を型検査時に判定することで，その関数定義が妥当かどうかを判断する．このとき，再帰呼び出しに関するサイズ減少の条件は本体の式から得られる制約に含まれる．

式

Emfrp$^{\mathrm{BCT}}$ の式について説明する．プリミティブな二項演算は Emfrp と同様のものである．ローカル変数定義も構文は異なるものの機能は同等である．以降では Emfrp$^{\mathrm{BCT}}$ において特徴的となる BCT を扱う式とそこから抽出される制約を列挙して解説する．

- 関数呼び出し f(e,...,e)では，引数に対する通常の型検査に加え，関数呼び出しの事前条件が制約として抽出される．さらに，f が再帰呼び出しだった場合には，定義時に指定された尺度関数にしたがって引数のサイズが単調に減少しているという制約が追加される．
- 結果の型が BCT となる `if`式では，`then`節と `else`節の結果の型の検査が行われるとともに，それらのサイズが等しいという制約が抽出される．
- 分解式 `case` e `of return` τ `of` $branch_1$|...|$branch_n$ではコンストラクタごとに BCT の分解を行う．$branch$ は対象の型の全てのコンストラクタを網羅していることが要求される．τ は結果の型を表す．`case`式からは全てのブランチの結果のサイズが τ と等しくなるという制約が抽出される．それぞれの $branch$ は χ(x1:π,...,xn:π) -> eの形である．ブランチではコンストラクタの保持する値を型注釈付きの変数名で束縛する．この時の型注釈は関数引数に使用できるものと同様で，新しくサイズ変数を導入することができる．また，使用しない型注釈は省略できる．新しく導入するサイズ変数には分解する式のサイズと関係した条件が課される．
- サイズ拡張式 e `adj[`ψ`]`は，BCT のサイズを大きい方へと調整する式である．BCT のサイズは最大要素数を表すため，サイズの小さな値をサイズの大きな値だと見なすことは安全である．制約として (e のサイズ $\leq \psi$) が抽出される．
- サイズキャスト式 `fit` e_0 `to` x:ρ^k -> e_1 | `fail` -> e_2は e_0 の実行時の真のサイズによって処理を分岐する式である．Emfrp$^{\mathrm{BCT}}$ では BCT の値は実行時のサイズ情報を持つ．そのサイズが定数サイズ k 以下であるならば値のサイズをキャストする．キャストされた値は x に束縛され，e_1 で使用できる．そうでないならば `fail`節の式を実行する．この式は主にノード定義式中にノードの直前値とともに使うことが想定されている．

上記の分解式 (`case`) について，前述の `TreeL`型を用いてブランチから抽出されるサイズ制約を説明する．`case`式にて `TreeL[5]`型の値を分解するとき，`NodeL`のブランチは `NodeL(l: TreeL[a],v: ListI[b], r: TreeL[c]) ->` eの形となる．このとき新しく導入したサイズ変数には $(a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (1 + a + c = 5) \wedge (b = 10)$ という条件が課される．結果としては，e から得られるサイズ制約を C とすると，このブランチからは $\forall a, b, c.((a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (1 + a + c = 5) \wedge (b = 10) \rightarrow (C \wedge$ 結果のサイズに関する制約)) という制約が抽出される．BCT の値が生成される際には，どのようなサイズの BCT 同士が合成されているかという情報が欠落してしまうため，`case`式のサイズ制約にて，その情報を復元している．

ノード定義

ノードは Emfrp と同様に定義する．ノード定義にて，新しいサイズ変数は `case`ブランチでのみ導入される．そのため，ノード定義式中の `case`ブランチ外では，サイズパラメータは定数が指定される．一方で，関数定義の際には `case`ブランチでの導入に加えて，引数から新しいサイズ変数が導入されうることに注意されたい．

使用メモリ量の決定

Emfrp では構文木を走査して，定数値やコンストラクタ適用などの値を生成する式を数えることで実行時に必要なメモリ量を決定する．元来の Emfrp では関数の再帰呼び出しが存在しないため，関数呼び出しではその関数の本体の構文木を展開することでノード更新全体のプログラムを走査できる．

Emfrp$^{\mathrm{BCT}}$ でも，Emfrp と同様に構文木を走査して値の生成を追跡することでメモリ量を決定する．使用メモリ量の決定は型検査，制約検査を通過したモジュールに対して行われる．式の構文木を走査する際には，式中に現れるサイズ変数に具体的な値を割り当てていく．関数呼び出しの際には導入されるサイズ変数の具体的な値を計算し，それをサイズ変数に割り当てて構文木を展開する．再帰呼び出しに

よって同じ関数が展開されることがありうるが，サイズの単調減少が保証されているため構文木の展開は有限回数で停止する．

`case`式では，ブランチをサイズについて網羅的に走査する．ブランチでは分解する型のサイズに関する制約の下で，サイズ変数が導入される．その制約を充足する付値の全ての組み合わせを導入されるサイズ変数に対して割り当ててブランチの式を走査する．BCT の値を分解する際には，**どのようなサイズの BCT 同士が合成されているか**という情報が欠落しているため，**制約を満たすサイズの組を全て列挙**することで情報を復元する．また，制約を充足できるような割り当てが存在しない場合にはそのブランチは走査しない．

最後に，ノード定義式を起点としてサイズ変数の具体的な値を定めながら構文木の走査を行い，ノード更新で必要とされるメモリ量を計算する．モジュールの実行をし続けるために必要なメモリ量は以下の 4 つの合計値である．

- それぞれのノードの現在値のためのメモリ量
- それぞれのノードの直前値のためのメモリ量
- それぞれのノードを更新するために必要なメモリ量のうち，最も大きいもの
- 言語ランタイムが必要とするメモリ量 (ガベージコレクションのための一時的な領域など)

3.3.4 記述例

第 3.2 章で提示した二つのモジュールを $\text{Emfrp}^{\text{BCT}}$ で記述した．それらの例を用いて，抽出されるサイズ制約や使用メモリ量決定のための構文木の走査について解説する．

DupCheck モジュール

図 3.4 に $\text{Emfrp}^{\text{BCT}}$ で記述した DupCheck モジュールを示す．モジュールの入力は Emfrp による例と同様である．6 行目で `Int`型を要素にもつリスト型の BCT を定義している．8–22 行目はリストのための補助関数の定義である．24–29 行目では `List[5]`型の履歴を表すノード `history`を定義している．履歴を 4 つ保持するために，リスト末尾の `Nil`と合わせてサイズが **5** に設定されていることに注意されたい．ノード定義式中では `fit`式を用いてリストの実行時サイズを取得し，その長さによって処理を分岐している．履歴のサイズに余裕がある場合は単純に入力値を履歴リストに追加する．そうでない場合には先頭を取り除き，入力値を末部に追加している．31 行目は履歴の直前値に現在入力された値が存在するかを検査するノード `detect`の定義である．

リストを用いて履歴を表現することで，補助関数が履歴長によらず共通に使用できるため，タプルを用いた記述と比べて保守性に富んだコードになっている．また，履歴長の書き換えの際にサイズについての整合性が静的に検査されるため，サイズに関する書き換えミスが検出できる．

`insert`関数を例として，再帰関数定義の妥当性の検査の際に抽出される制約を説明する．まず，`insert`関数の引数から，新しいサイズ変数 `m`が導入される．この関数の呼び出しの際には，事前条件として $m > 0$ が要請される．

関数定義式の制約はボトムアップに抽出される．`Nil`のブランチからは，コンストラクタ適用，`adj`式，結果のサイズについての制約から $\top \to 2 \leq m + 1 \wedge m + 1 = m + 1$ が抽出される．これを X とする．次に，`Cons`ブランチではサイズ m の変数 `l`を分解するため，新しいサイズ変数 `n`が条件 $n > 0 \wedge m = 1 + n$ のもとで導入される．後続の式の再帰呼び出し `insert(x, t)`に注目すると，関数呼び出しの前提条件 $(n > 0)$ と再帰呼び出しに関する制約 $(n < m)$ の連言が抽出される．再帰呼び出しに関する制約は，9 行目末の尺度関数の記述 `[m]`により指定されている (BCT 型の実引数のサイズ合計が `m`よりも小さい制約)．つまりこの再帰呼び出しからは，$n > 0 \wedge n < m$ が抽出さ

```
1  module DupCheck
2  in  reset  : Bool init (False), # reset signal
3      v      : Int  init (0)      # input value
4  out detect : Bool
5
6  type List = Nil | Cons(Int, List)
7
8  func insert(x: Int, l: List[m]): List[m+1] where {m > 0} [m] =
9    case l return List[m+1] of
10   | Nil -> Cons(x, Nil) adj[m+1]
11   | Cons(h:Int,t:List[n]) -> Cons(h,insert(x,t))
12
13 func search(x: Int, l: List[m]): Bool where {m > 0} [m] =
14   case l return Bool of
15   | Nil -> False
16   | Cons(h: Int, t: List[n]) ->
17       if x = h then True else search(x,t)
18
19 func tail(l: List[m]): List[m-1] where {m-1 > 0} =
20   case l return List[m-1] of
21   | Nil -> Nil adj[m-1]
22   | Cons(h: Int, t: List[n]) -> t
23
24 node history: List[5] init (Nil adj[5]) =
25   if reset then Cons(v, Nil) adj[5]
26   else
27     fit history@last to
28     | hl: List[4] -> insert(v,hl)
29     | fail -> insert(v,tail(history@last))
30
31 node detect: Bool init (False) = search(v, history@last)
```

図 3.4　$\mathrm{Emfrp}^{\mathrm{BCT}}$ による DupCheckモジュール

れる．再帰呼び出しの結果をコンストラクタ適用したサイズとブランチの結果のサイズに注意すると，制約 $1 + (n + 1) = m + 1$ が得られる．したがって，Consブランチ全体から抽出される制約は $\forall n.(n > 0 \land m = 1 + n) \to (n > 0 \land n < m \land 1 + (n + 1) = m + 1)$ である．これを Y とする．

以上の制約と関数の結果のサイズが $m + 1$ となる要請に注意すると，関数の妥当性検査で検査すべきサイズ制約は $\forall m.(m > 0) \to (X \land Y \land m + 1 = m + 1)$ となる．ここで示した制約は実際の型検査の際に抽出される制約ではないことに注意されたい．実際の制約は変数参照などに対し恒真 ($\top$) な制約を追加するため，ここで示した制約よりも煩雑なものとなるが，本質的には同等である．他の関数定義やノード定義についても同様の型検査，制約抽出が行われ定義の妥当性が判断される．

ノード更新に必要なメモリ使用量決定の際には，同じ構文木 (特に caseのブランチ) に対する走査が複数回行われることがある．ただし，この例のようにリスト構造を扱う場合には，新しく導入されるサイズ変数の大きさが一意に定まる (m が定まっている時，条件 $m = 1 + n$ を満たす n は一意に定まる) ため，caseのブランチの探索はそれぞれ一度だけ行われる．例えば insert関数の場合にはサイズ変数に $m = 4, 3, 2, 1$ が順に割り当てられて定義式が走査されることになる．

Top10Sum モジュール

図 3.5 に $\mathrm{Emfrp}^{\mathrm{BCT}}$ で記述した Top10Sum モジュールを示す．以前の例では値を管理するためにタプルを用いていたが，ここでは左偏ヒープ [49] を用いている．モジュールの入出力は Emfrp の例と同様である．8–9 行目でヒープ木の型を定義している．コンストラクタ Tのパラメータの 3 つ目と 4 つ目が再帰的になっている．11–36 行目では左偏ヒープの補助関数を定義している．39–43 行目で

```
# The sum of the top 10 of the input data
module Top10Sum
in  reset: Bool init (False),
    x:     Int init (0)      # input value
out sum: Int    # sum

# Leftist Heap
type Heap = E                # terminal
     | T(Int,Int,Heap,Heap) # (rank,v,left,right)

func rank(e: Heap[n]) : Int where {n > 0} =
  case e return Int of E -> 0 | T(r, x, a, b) -> r

func make(x:Int,a:Heap[n],b:Heap[m]): Heap[1+n+m] where {n > 0, m > 0} =
  let ra = rank(a) in let rb = rank(b) in
  if ra > rb then T(rb+1,x,a,b) else T(ra+1,x,b,a)

func merge(h1:Heap[n],h2: Heap[m]): Heap[n+m-1] where {n > 0, m > 0} [n,m] =
  case h1 return Heap[n+m-1] of
  | E -> h2 adj[n+m-1]
  | T(r1, x1, a1, b1) ->
    case h2 return Heap[n+m-1] of
    | E -> h1 adj[n+m-1]
    | T(r2, x2, a2, b2) ->
      if x1 <= x2 then make(x1, a1, merge(b1, h2))
        else make(x2, a2, merge(h1, b2))

func insert(x: Int, h: Heap[n]): Heap[n+2] where {n > 0} = merge(T(1, x, E, E), h)

func findMin(h: Heap[n]): Int where {n>0} =
  case h return Int of E -> 0 | T(r, x, a, b) -> x

func delMin(h:Heap[n]):Heap[n-2] where {n>2} =
  case h return Heap[n-2] of
  | E -> E adj[n-2]
  | T(r, x, a, b) -> merge(a, b)


node h: Heap[21] init (E adj[21]) =
  if reset then E adj[21] else
  fit h@last to hl: Heap[19] -> insert(x, hl)
  | fail -> if x <= findMin(h@last) then h@last
              else insert(x, delMin(h@last))

node sum : Int init (0) = if reset then 0 else
  let diff = fit h@last to
     | hl : Heap[19] -> x
     | fail ->
        if x <= findMin(h@last) then 0
          else x - findMin(h@last))
  in sum@last + diff
```

図 3.5　Emfrp$^{\mathrm{BCT}}$ による Top10Sumモジュール

定義されているノード hはサイズ 21 のヒープ木である．これは 10 個の値を保持するためのサイズである．hはノード更新の際，要素数に余裕があれば入力値をヒープに挿入する．要素数に余裕がない場合には，ヒープに格納されている最小値を取得し入力値と比較する．入力値が最小値よりも大きい場合には，入力値は上位 10 個以内に含まれているので，その最小値をヒープから取り除き，代わりに入力値を挿入する．そうでない場合には現状を維持する．この時，ヒープへの挿入と最小値の削除は

$O(\log(\text{ヒープの要素数}))$，ヒープの最小値の取得は $O(1)$ の時間計算量で行うことができる．45–51 行目は合計値を計算するノード `sum`の定義である．ヒープに値が挿入される時に入力値とヒープの最小値の差分を計算し，自身の直前値に加算することで合計値を更新する．この更新処理は $O(1)$ で行うことができる．モジュール全体で見ると，ノードの更新には $O(\log(\text{ヒープの要素数}))$ の時間計算量が必要である．したがって，このプログラムは更新処理に線形時間かかる Emfrp の例よりも時間効率がよく，要素数を多くしても処理に時間がかかりづらいと言える．

このプログラムにおいて再帰関数は 18–26 行目で定義されている `merge`関数だけである．この関数は呼び出される際に，引数のうちどちらかのサイズのみが小さくなる．そのため，尺度関数を `[n,m]`にして，2 つの引数のサイズの合計が単調に減少することを検査する．この検査にパスすることで `merge`関数が停止することが保証される．

ノード更新時の使用メモリ量の決定の際，`case`式によるヒープ木の分解では `T`のブランチで木の構造による全探索が行われる．サイズ 21 の木を分解する場合，コンストラクタ `T`のパラメータの 3 つ目と 4 つ目のサイズ (m, n) には $21 = 1 + m + n$ の条件がある．したがって $(m, n) = (1, 19), (2, 18), (3, 17), \ldots, (18, 2), (19, 1)$ を順にサイズとして割り当て，同一の構文木 (特にブランチの後続式) が複数回走査されることになる．

提案する使用メモリ量の決定アルゴリズムは，データ型の取りうる内部構造 (サイズ割り当て) を全て列挙しながら探索を行うため，組合せ爆発を引き起こす可能性がある．$\text{Emfrp}^{\text{BCT}}$ は，小規模な組込みシステムを実行対象としたプログラムを記述することを想定しているため，提案アルゴリズムは比較的サイズの小さなデータ型を対象とすることが予想される．そのため，提案アルゴリズムの実行時間は多くの場合，実用的な範囲に収まると考えられるが，より規模の大きい対象に対する検討は将来課題である．

3.4 形式化

本節では $\text{Emfrp}^{\text{BCT}}$ の構文，操作的意味論，型システムの形式化を行う．その後，式についての使用メモリ量決定アルゴリズムを提示し，健全性，すなわちアルゴリズムから得られたメモリ量でノード更新が行えることを示す．

3.4.1 構文

図 3.6 に $\text{Emfrp}^{\text{BCT}}$ の構文を示す．ほとんどの構文は第 3.3 章で説明した $\text{Emfrp}^{\text{BCT}}$ の具象構文と同様である．具象構文と異なっているのは次の点である．

- サイズパラメータの位置が型名の上付き添字となっている点
- `case`式のブランチにおいて変数が導入される場合，必ず型注釈をつける必要がある点

また，モジュールにおいて出力ノードを明示的には宣言していない．

図中に示した構文規則の他に以下のような構文に関する要請がある．

- `case`式のブランチは分解する型の全てのコンストラクタが網羅されていること
- 関数定義内にノード参照や直前値参照が含まれていないこと
- 型名，コンストラクタ名，関数名，変数名，サイズ変数名はユニークに名前付けられていること
- ノード定義や関数定義が以下に定義する順序にしたがって整列していること

定義 1 (ノード定義の依存順序)**.** ノード N の更新式 ($\mathcal{N}(N)$ の 2 番目の要素) にノード N' への参照が含まれている，すなわち N が N' に依存するとき，$N >_{\mathcal{N}} N'$ と定義する．ノード N の依存関係に循

Size Constraints

$$
\begin{array}{llll}
k & \in & \mathbb{N} \qquad \delta \in \mathit{SizeVar} & \\
\mathcal{C} & \in & \mathit{Constraint} & ::= \quad \top \mid \mathcal{A} \mid \forall\delta.\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C}' \mid \mathcal{C} \to \mathcal{C}' \\
\mathcal{A} & \in & \mathit{ArithCon} & ::= \quad \psi = \psi' \mid \psi < \psi' \mid \psi \leq \psi' \\
\psi & \in & \mathit{SizeParam} & ::= \quad k \mid \delta \mid \psi + \psi' \mid \psi - \psi'
\end{array}
$$

Types

$$
\begin{array}{llll}
\mathcal{B} & \in & \mathit{BaseType} & ::= \quad \texttt{Bool} \mid \texttt{Int} \\
\rho & \in & \mathit{TypeName} & \\
\pi & \in & \mathit{VarType} & ::= \quad \mathcal{B} \mid \rho^{\delta} \\
\sigma & \in & \mathit{ConstType} & ::= \quad \mathcal{B} \mid \rho^{k} \\
T^{\rho} & \in & \mathit{ElemType} & ::= \quad \mathcal{B} \mid \rho \mid \rho'^{k} \quad (\rho' \neq \rho) \\
\tau & \in & \mathit{Type} & ::= \quad \mathcal{B} \mid \rho^{\psi}
\end{array}
$$

Expressions

$$
\begin{array}{llll}
i & \in & \mathit{Integer} \qquad x \in \mathit{Var} \qquad N \in \mathit{Node} & \\
\chi & \in & \mathit{ConstructorLabel} \qquad f, g \in \mathit{Function} & \\
c & \in & \mathit{Constant} ::= \texttt{true} \mid \texttt{false} \mid i & \\
e & \in & \mathit{Expr} & \\
 & ::= & c^{\mathcal{B}} \mid x \mid N \mid N\texttt{@last} \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e & \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ op^{(\mathcal{B}_1,\mathcal{B}_2)\to\mathcal{B}}\ e & \\
 & \mid & f\texttt{(}\overrightarrow{e}\texttt{)} \mid \chi\texttt{(}\overrightarrow{e}\texttt{)} \mid \texttt{case}\ e\ \texttt{return}\ \tau\ \texttt{of}\ \overrightarrow{branch} & \\
 & \mid & e\ \texttt{adj [}\psi\texttt{]} \mid \texttt{fit}\ e\ \texttt{to}\ x\texttt{:}\rho^{k} \to e \mid \texttt{fail} \to e & \\
\mathit{branch} & \in & \mathit{Branch} ::= \chi\texttt{(}\overrightarrow{x\texttt{:}\pi}\texttt{)} \to e & \\
\mathit{ce} & \in & \mathit{ConstExpr} \subset \mathit{Expr} &
\end{array}
$$

Module Definitions

$$
\begin{array}{llll}
M & \in & \mathit{Module} ::= (\mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{N}, \mathit{Init}) & \\
\mathcal{I} & ::= & \overrightarrow{N : \sigma} & \text{(Input Nodes)} \\
\mathcal{T} & ::= & \cdot \mid \mathcal{T}, \rho \mapsto \chi(\overrightarrow{T^{\rho}}) & \text{(Type Definitions)} \\
\mathcal{N} & ::= & \cdot \mid \mathcal{N}, N \mapsto (\sigma, e) & \text{(Node Definitions)} \\
\mathcal{F} & ::= & \cdot \mid \mathcal{F}, f \mapsto \texttt{(}\overrightarrow{x\texttt{:}\pi}\texttt{)}\texttt{:}\tau\ \texttt{where}\ \texttt{\{}\overrightarrow{\mathcal{A}}\texttt{\}[}\overrightarrow{\delta}\texttt{]}\ \texttt{=}\ e & \\
 & & & \text{(Function Definitions)} \\
\mathit{Init} & ::= & \cdot \mid \mathit{Init}, N \mapsto \mathit{ce} & \text{(Initial Values)}
\end{array}
$$

図 3.6　Emfrp$^{\mathrm{BCT}}$ の構文

環が存在しないため，$>_{\mathcal{N}}$ の推移閉包 $>^{*}_{\mathcal{N}}$ は狭義半順序関係となる．

定義 2 (関数定義の依存順序)**.** 関数 $f \in dom(\mathcal{F})$ の定義式に関数 g の呼び出しが含まれている，すなわち f が g に依存するときあるいは $g = f$ のとき，$f \geq_{\mathcal{F}} g$ と定義する．関数 f の定義中に f の呼び出しは可能であり，関数の相互再帰は禁止されているため，$\geq_{\mathcal{F}}$ の推移閉包 $\geq^{*}_{\mathcal{F}}$ は半順序関係となる．

サイズ制約の構文について説明する．k はサイズ定数，δ はサイズ変数を表す．サイズ変数はサイズパラメータ上の変数であり，正の整数の範囲をとる．$\mathcal{C}$ は型検査時に抽出されるサイズに関する制約である．真のリテラル，算術制約，サイズ変数による全称量化，制約の連言，制約の含意から構成される．$\mathcal{A}$ は関数定義の際に指定する関数呼び出しのための事前条件，ψ は BCT に設定できるサイズパラメーターである．

次に型について説明する．$\mathcal{B}$ は基本型であり，整数型とブール型がある．ρ は BCT の名前に使用される．出現位置によって許される型の形が異なるためいくつかのバリエーション (π，σ，T^{ρ}，τ) が用意されている．特に T^{ρ} はコンストラクタのパラメータで使用できる型で，再帰している (自己言及されている) 型をサイズパラメータを伴わない型名 ρ で表すことができる．

式については以下の注意点がある．

Values
$l \in \mathit{Location}$
$v \in \mathit{Value} \quad ::= c^{\mathcal{B}} \mid \chi[k](\overrightarrow{l})$
Environments
$E ::= \cdot \mid E, x \mapsto l$ (Local Var. Env.)
$H ::= \cdot \mid H, l \mapsto v$ (Heap)
$\mathcal{L} ::= \cdot \mid \mathcal{L}, N \mapsto l \mid \mathcal{L}, N\texttt{@last} \mapsto l$ (Node Locations)
$\Gamma ::= \cdot \mid \Gamma, N : \tau \mid \Gamma, x : \tau$ (Type Env.)
$\Delta ::= \cdot \mid \Delta, \delta \mapsto k$ (Size Var. Env.)
Recursion Indices
$I(\chi) = \{i \mid i \in 1 \ldots n, T_i^\rho = \rho\}$
where $\chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho)$

図 3.7　値の定義，環境の定義，再帰位置のための補助関数の定義

- `case`式では結果の型を明示する必要がある．
- ネストしたパターンマッチについては扱わない．
- ce はノードの初期値などに指定される定数式を表す．図 3.6 では明記していないが，整数値やコンストラクタの呼び出しのみが許される式である．

モジュールは型定義の列 $\mathcal{T}$，関数定義の列 $\mathcal{F}$，入力ノードの列 $\mathcal{I}$，中間または出力ノードの列 $\mathcal{N}$，それぞれのノードの初期値の列 Init の 5 つ組によって構成される．具象構文で `type`をもちいて定義した BCT は型名からコンストラクタ定義への辞書 ($\mathcal{T}$) に格納されている．

3.4.2　操作的意味論

図 3.7 に値の定義，環境の定義，再帰位置のための補助関数の定義を示す．l はヒープ領域の位置を表す．v は値を表し，整数やブール値の定数か BCT の値を表す．BCT の値はコンストラクタのタグ，実行時のサイズ情報，パラメータを表す位置の列を持つ．E はローカル変数名から位置への環境，H はヒープ領域での位置から値への環境，$\mathcal{L}$ はノード名からヒープの位置への環境である．ノードの直前値参照も $\mathcal{L}$ から参照される．Γ は型環境であり，型検査時に用いられる．Δ はサイズ変数からサイズ定数への環境であり，使用メモリ決定の際に用いられる．補助関数 $I(\chi)$ は，コンストラクタ χ のパラメータ上で再帰している型が現れる位置 (添字) の集合である．

図 3.8 に式の大ステップ操作的意味論を示す．Emfrp$^{\mathrm{BCT}}$ は計算中のリソース使用量に注目した言語であるから，意味論中では要求されるリソースのいくつかを明示している．明示されるリソースは，参照するローカル変数の数，ヒープにストアされるデータの数，関数呼び出しの回数である．操作的意味論は $[s]E \mid [t]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l'; [t']H'$ によって表示される．これは「型定義 $\mathcal{T}$，関数定義 $\mathcal{F}$，ノード位置環境 $\mathcal{L}$ において，空き容量 s のローカル変数環境 E，空き容量 t のヒープ H，コールスタックの空き容量 u の下で式 e を評価すると，評価後に空き容量 t' のヒープ H' が得られ，評価結果は H' 中の位置 l' にストアされる」と読む．式の評価結果が値を直接表すのではなく，評価後のヒープ中の位置を表すことに注意されたい．ノードを評価する際に渡す $\mathcal{L}$ の定義域 (ドメイン) は $\mathcal{N}$ と $\mathcal{I}$ の定義域の和集合である．Emfrp では，関数内部ではノードの現在値や直前値を参照することができないという制限がある．Emfrp$^{\mathrm{BCT}}$ でもこの制限を反映するために，関数の評価 (規則 E-CALL) ではノード位置環境 $\mathcal{L}$ を空を表す $\cdot$ に変更していることに注意されたい．

E はローカル変数のためのスタックメモリ領域を表し，その空き容量が s で与えられる．同様に H, H' は値をストアするためのヒープメモリ領域を表し，その空き容量が t, t' によって与えられている．また，u はコールスタックの空き容量，つまり関数呼び出しの回数を示す．それぞれの空き容量

$\boxed{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l; [t']H'}$

$$\frac{l \notin dom(H)}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} c^{\mathcal{B}} \Downarrow_u l; [t-1](H, l \mapsto c)} \text{(E-CONST)} \qquad \frac{E(x) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} x \Downarrow_u l; [t]H} \text{(E-VAR)}$$

$$\frac{\mathcal{L}(N) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} N \Downarrow_u l; [t]H} \text{(E-NODE)} \qquad \frac{\mathcal{L}(N\texttt{@last}) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} N\texttt{@last} \Downarrow_u l; [t]H} \text{(E-ATLAST)}$$

$$\frac{\left[\ [s-1]E \mid [t_{i-1}]H_{i-1} \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_i \Downarrow_u l_i; [t_i]H_i\ \right]_{i \in 1 \ldots n} \qquad \chi(T^{\rho}_1, \ldots, T^{\rho}_n) \in \mathcal{T}(\rho) \qquad [H_n(l_i) = \chi'_i[k_i](\ldots)]_{i \in I(\chi)} \qquad k = 1 + \textstyle\sum_{i \in I(\chi)} k_i \qquad l \notin dom(H_n)}{[s]E \mid [t_0+1]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \chi(e_1, \ldots, e_n) \Downarrow_u l; [t_n](H_n, l \mapsto \chi[k](l_1, \ldots, l_n))} \text{(E-CTOR)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \chi_a[k](l'_1, \ldots, l'_{ar_a}) \qquad 1 \le a \le n \qquad [s - r_a](E, x_{a1} \mapsto l'_1, \ldots, x_{ar_a} \mapsto l'_{ar_a}) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_a \Downarrow_u l_2; [t_2]H_2}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{case}\ e\ \texttt{return}\ \tau\ \texttt{of}\ \{\chi_i(x_{i1}\texttt{:}\tau_{i1}, \ldots, x_{ir_i}\texttt{:}\tau_{ir_i}) \to e_i\}_{i \in 1 \ldots n} \Downarrow_u l_2; [t_2]H_2} \text{(E-CASE)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad [s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow_u l_2; [t_2]H_2} \text{(E-LET)}$$

$$\frac{\left[\ [s-(i-1)]E \mid [t_{i-1}]H_{i-1} \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_i \Downarrow_u l_i; [t_i]H_i\ \right]_{i \in 1 \ldots n} \qquad \mathcal{F}(f) = (x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n)\texttt{:}\tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}\texttt{[}\overrightarrow{\delta}\texttt{]} = e \qquad [s-n](x_1 \mapsto l_1, \ldots, x_n \mapsto l_n) \mid [t_n]H_n \vdash^{\mathcal{T};\mathcal{F}} e \Downarrow_{u-1} l; [t']H'}{[s]E \mid [t_0]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} f(e_1, \ldots, e_n) \Downarrow_u l; [t']H'} \text{(E-CALL)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \texttt{true} \qquad [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_2; [t_2]H_2} \text{(E-IF-THEN)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \texttt{false} \qquad [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_3 \Downarrow_u l_3; [t_3]H_3}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_3; [t_3]H_3} \text{(E-IF-ELSE)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad [s-1]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2 \qquad H_2(l_1) = v_1 \qquad H_2(l_2) = v_2 \qquad v = op(v_1, v_2) \qquad l \notin dom(H_2)}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1\ op^{(\mathcal{B}_1, \mathcal{B}_2) \to \mathcal{B}}\ e_2 \Downarrow_u l; [t_2 - 1](H_2, l \mapsto v)} \text{(E-OP)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l; [t']H'}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e\ \texttt{adj[}\psi\texttt{]} \Downarrow_u l; [t']H'} \text{(E-ADJ)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \chi[k'](\overrightarrow{l'}) \qquad k' \le k \qquad [s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{fit}\ e_1\ \texttt{to}\ x\texttt{:}\rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3 \Downarrow_u l_2; [t_2]H_2} \text{(E-FIT-SUCCESS)}$$

$$\frac{[s]E \mid [s]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \chi[k'](\overrightarrow{l'}) \qquad k' > k \qquad [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_3 \Downarrow_u l_3; [t_3]H_3}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{fit}\ e_1\ \texttt{to}\ x\texttt{:}\rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3 \Downarrow_u l_3; [t_3]H_3} \text{(E-FIT-FAIL)}$$

図 3.8　式の操作的意味論

s, t, t', u は常に 0 以上であることが要求される．評価の途中にいずれかの空き容量が負になる時には，その評価はスタック (stuck，異常終了) する．加えて，環境への参照などの補助関数として使用している部分関数が，定義域に含まれていない値に適用される場合には評価はスタックする．

式の評価中，値はヒープにストアされるが，一度ヒープにストアされた値やストアした位置は変更されない．つまり，式の評価の途中でガベージコレクションなどのヒープの空き容量を増やす操作は行われない．そのため，式の評価では，評価後のヒープ H' は評価前のヒープ H を含み ($H \subseteq H'$)，評価後のヒープの空き容量 t' は評価前のヒープの空き容量 t と同じか小さくなる ($t \geq t'$)．

E には局所的に記憶する必要のある位置を保持する．規則 E-CASE，E-LET，E-FIT-SUCCESS では新しくローカル変数が導入されるため，後続の式の評価の際に E に束縛を追加しその空き容量を減少させる．規則 E-CALL，E-OP は，引数を評価したのちに関数や二項演算を適用する規則である．このとき，引数の評価結果の位置を一時的に記憶しておく必要がある．この振る舞いを，「他の式の評価時に参照されない変数がローカル変数環境に束縛される」と解釈し，二番目以降の引数の評価の際にローカル変数環境を E-CALL では $[s-(i-1)]E$，E-OP では $[s-1]E$ として空き容量を減らすことで表現する．規則 E-CTOR では，結果を保存する位置があらかじめヒープ上に確保される．その後，確保された位置の情報をローカル変数へ追加したのちに，コンストラクタのパラメータの評価を行う．これらのパラメータの評価結果の位置はヒープに確保した位置のパラメータ部分に順次記述されると解釈する．この振る舞いを表現するために，入力ヒープ領域の空き容量が $[t_0+1]$ となっている．[*2]

他の規則は典型的な値呼びの関数型言語の評価規則と同様である．評価中には `fit`式を除きプログラムに記述されたサイズパラメータが扱われることはない．`fit`式ではオブジェクトの実行時のサイズ情報を用いて評価の分岐が行われるが，定数との比較のみ行われるようになっており，サイズ変数は出現しない．

3.4.3 型システム

図 3.9 に型付け規則で用いる補助演算子の定義を示す．$\cdot \sim \cdot$ は型を単一化するために必要なサイズ制約を生成する演算子である．両辺が基本型同士の場合には恒真の制約，両辺が BCT の場合にはサイズパラメータが等しいという制約が生成される．最後のパターンは，コンストラクタに現れる再帰している型 (のプレースホルダ) のための制約生成を行う．$\tau\downarrow$ は型からサイズパラメータを抽出する演算子である．3 つ目の規則群はサイズパラメータの代入規則である．最後の規則は，型の列の組からサイズパラメータ代入を生成する規則である．対応した型同士をサイズについて単一化し，得られた制約が $\delta = \psi$ の形を持つ場合に，サイズパラメータ代入 $\delta \mapsto \psi$ を生成する．π 形の型はプリミティブ型も含むため，単一化で得られた制約が $\delta = \psi$ の形を持たない場合があることに注意されたい．

図 3.10 に式の型付け規則を示す．型付け規則は型判断 $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C}$ で表される．これは，「型定義 $\mathcal{T}$，関数定義 $\mathcal{F}$ において，型環境 Γ の下で関数 f の内部式 e は型 τ を持ち，サイズ制約 $\mathcal{C}$ が抽出される」と読む．また，`case`式のブランチのための型付け規則は型判断 $\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} branch :\sim \tau \mid \mathcal{C}$ で表される．これは「型定義 $\mathcal{T}$，関数定義 $\mathcal{F}$ において，型環境 Γ の下で関数 f の内部式のサイズ ψ の式を分解するブランチ $branch$ は型 τ を持ち，サイズ制約 $\mathcal{C}$ が抽出される」と読む．それぞれの型判断では，どの関数の型検査を行っているかを表す関数名 f が渡されている．ノード定義式や，初期値の型付けの際には空を表す特殊な名前 $-$ が渡される．

型検査と同時に抽出されるサイズ制約 $\mathcal{C}$ は 3.4.1 節で提示したように，正の整数に対する加減算と比較，変数による全称量化，論理式同士の連言，含意によって構成されている．これはプレスバーガー算術として知られた論理体系の範疇に含まれる．プレスバーガー算術では論理式の真偽判定が計算可能

[*2] 文献 [77] での規則 E-CTOR とは異なる定義となっている．処理系の振る舞いをより厳密に模倣するために本章での定義へと修正した．なおこの修正による健全性の証明への影響はない．

$\boxed{\tau \sim \tau' \qquad T^{\rho} \sim \tau'}$

$$
\begin{aligned}
\mathcal{B} \sim \mathcal{B} &= \top \\
\rho^{\psi} \sim \rho^{\psi'} &= (\psi = \psi') \\
\rho \sim \rho^{\psi} &= (\psi > 0)
\end{aligned}
$$

$\boxed{\tau\downarrow}$

$$
\begin{aligned}
\mathcal{B}\downarrow &= 0 \\
\rho^{\psi}\downarrow &= \psi
\end{aligned}
$$

$\boxed{[\overrightarrow{\delta \mapsto \psi}]\tau \qquad [\overrightarrow{\delta \mapsto \psi}]\mathcal{A} \qquad [\overrightarrow{\delta \mapsto \psi}]\psi}$

$$
\begin{aligned}
[\overrightarrow{\delta \mapsto \psi}]\mathcal{B} &= \mathcal{B} \\
[\overrightarrow{\delta \mapsto \psi}]\rho^{\psi'} &= \rho^{[\overrightarrow{\delta \mapsto \psi}]\psi'} \\
[\overrightarrow{\delta \mapsto \psi}](\psi \circ \psi') &= [\overrightarrow{\delta \mapsto \psi}]\psi \circ [\overrightarrow{\delta \mapsto \psi}]\psi' \\
&\qquad (\text{if } \circ \in \{+, -, =, <, \leq\}) \\
[\overrightarrow{\delta \mapsto \psi}]k &= k \\
[\delta_1 \mapsto \psi_1, \ldots, \delta_n \mapsto \psi_n]\delta &= \delta \quad (\text{if } \delta \neq \delta_i \text{ for any } i) \\
[\delta_1 \mapsto \psi_1, \ldots, \delta_n \mapsto \psi_n]\delta &= \psi_i \quad (\text{if } \delta = \delta_i)
\end{aligned}
$$

$\boxed{[\overrightarrow{\pi} \mapsto \overrightarrow{\tau}]}$

$$
\begin{aligned}
&[\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n] \quad = \\
&\qquad [\{\, \delta \mapsto \psi \mid 1 \leq i \leq n, \pi_i \sim \tau_i = (\delta = \psi) \,\}] \\
&\qquad\qquad (\forall i \in 1 \ldots n, \pi_i \sim \tau_i \text{ is defined})
\end{aligned}
$$

図 3.9　型とサイズ制約のための補助演算子．型の単一化 ($\sim$)，サイズパラメータ抽出 ($\downarrow$)，型上のサイズパラメータ代入

であることが知られている．したがって型付けによって得られるサイズ制約の真偽判定も計算可能である．

推論規則のいくつかを説明する．規則 T-CTOR では，コンストラクタのパラメータの型それぞれについて型付けしたのち，再帰している型についての情報を使って結果のサイズを計算する．

規則 T-CASE では，分解対象の式から得られる制約と各ブランチから得られた制約を連言にしている．各ブランチは規則 T-BRANCH によって検査される．コンストラクタのパラメータのそれぞれからサイズ変数を抽出し，制約 $\mathcal{C}$ を条件として新たに導入する．制約 $\mathcal{C}$ は分解している型のサイズに由来する制約 $\mathcal{R}$ とその他の BCT のサイズのための制約の連言によって構成される．これらの制約と式から得られた制約 $\mathcal{C}'$ と結果のサイズに関する制約 $\tau' \sim \tau$ の連言を用いて最終的な制約を生成する．

規則 T-CALL では関数呼び出しに関する型検査と制約の抽出を行う．型検査に関しては一般的な一階の関数の関数呼び出しの型検査を行う．制約抽出では，まず $\mathcal{F}$ から得られた関数定義の仮引数のサイズ変数と実引数のサイズパラメータからサイズパラメータ代入 θ を生成する．関数呼び出しの事前条件 $\mathcal{A}$ は θ によって適切に置換される．関数が再帰呼び出しだった ($f = g$) 場合には，尺度関数を θ で置換したサイズ (再帰呼び出しのときのサイズ) が仮引数の尺度 (呼び出された時のサイズ) よりも減少しているという制約 $\mathcal{R}$ を生成する．この制約が追加されることで再帰関数呼び出しの停止性が保証される．

規則 T-IF では，`then`節と `else`節の結果のサイズが一致する ($\tau \sim \tau'$) という制約を追加している．

図 3.11 にモジュール定義の妥当性を判断するためのサイズ制約生成規則を示す．これらの規則では，関数定義，ノード定義，ノードの初期値について，それらの定義式から得られるサイズ制約を基に検査すべき制約を生成する．例えば関数定義の場合，一つ一つの関数ごとに全称量化された含意から構成さ

$\boxed{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C}}$

$$\frac{}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} c^{\mathcal{B}} : \mathcal{B} \mid \top}\ \text{(T-CONST)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} x : \tau \mid \top}\ \text{(T-VAR)} \qquad \frac{\Gamma(N) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} N : \tau \mid \top}\ \text{(T-NODE)} \qquad \frac{\Gamma(N) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} N\texttt{@last} : \tau \mid \top}\ \text{(T-ATLAST)}$$

$$\frac{\chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \qquad \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_i : \tau_i \mid \mathcal{C}_i\ \right]_{i \in 1 \ldots n} \qquad \psi = 1 + \textstyle\sum_{i \in I(\chi)} \tau_i \downarrow}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \chi(e_1, \ldots, e_n) : \rho^\psi \mid \bigwedge_{i \in 1 \ldots n} (\mathcal{C}_i \wedge T_i^\rho \sim \tau_i)}\ \text{(T-CTOR)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_0 : \rho^\psi \mid \mathcal{C}_0 \qquad \left[\ \Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} branch_i :\sim \tau \mid \mathcal{C}_i\ \right]_{i \in 1 \ldots n}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{case}\ e_0\ \texttt{return}\ \tau\ \texttt{of}\ \{branch_i\}_{i \in 1 \ldots n} : \tau \mid \bigwedge_{i \in 0 \ldots n} \mathcal{C}_i}\ \text{(T-CASE)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \tau_1 \mid \mathcal{C} \qquad \Gamma, (x : \tau_1) \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau_2 \mid \mathcal{C}'}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \mid \mathcal{C} \wedge \mathcal{C}'}\ \text{(T-LET)}$$

$$\frac{\begin{array}{c}\mathcal{F}(g) = (x_1 : \pi_1, \ldots, x_n : \pi_n) : \tau\ \texttt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}[\overrightarrow{\delta}] = e \\ \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_i : \tau_i \mid \mathcal{C}_i\ \right]_{i \in 1 \ldots n} \qquad \mathcal{R} = \begin{cases} \sum_{\delta \in \overrightarrow{\delta}} (\theta\delta) < \sum_{\delta \in \overrightarrow{\delta}} \delta & (f = g) \\ \top & \text{(otherwise)} \end{cases} \\ \theta = [\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n]\end{array}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} g(e_1, \ldots, e_n) : \theta\tau \mid \bigwedge_{i \in 1 \ldots n} \mathcal{C}_i \wedge \bigwedge_{i \in 1 \ldots m} (\theta\mathcal{A}_i) \wedge \mathcal{R}}\ \text{(T-CALL)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \texttt{Bool} \mid \mathcal{C}_1 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid \mathcal{C}_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid \mathcal{C}_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \tau \sim \tau' \wedge \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3}\ \text{(T-IF)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \mathcal{B}_1 \mid \mathcal{C}_1 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \mathcal{B}_2 \mid \mathcal{C}_2}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1\ op^{(\mathcal{B}_1, \mathcal{B}_2) \to \mathcal{B}}\ e_2 : \mathcal{B} \mid \mathcal{C}_1 \wedge \mathcal{C}_2}\ \text{(T-OP)} \qquad \frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \rho^{\psi'} \mid \mathcal{C}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e\ \texttt{adj}\ [\psi] : \rho^\psi \mid \mathcal{C} \wedge \psi' \leq \psi}\ \text{(T-ADJ)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \rho^\psi \mid \mathcal{C}_1 \qquad \Gamma, (x : \rho^k) \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid \mathcal{C}_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid \mathcal{C}_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{fit}\ e_1\ \texttt{to}\ x : \rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3 : \tau \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \tau \sim \tau'}\ \text{(T-FIT)}$$

$\boxed{\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} branch :\sim \tau \mid \mathcal{C}}$

$$\frac{\begin{array}{c}\chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \qquad \mathcal{R} = \begin{cases} \psi = 1 + \sum_{i \in I(\chi)} \pi_i \downarrow & (I(\chi) \neq \emptyset) \\ \top & \text{(otherwise)} \end{cases} \\ \overrightarrow{\delta} = \{\pi_i \downarrow \mid i \in 1 \ldots n, \pi_i \downarrow \in SizeVar\} \\ \Gamma, x_1 : \pi_1, \ldots, x_n : \pi_n \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau' \mid \mathcal{C}' \qquad \mathcal{C} = \mathcal{R} \wedge \bigwedge_{i \in 1 \ldots n} T_i^\rho \sim \pi_i\end{array}}{\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} \chi(x_1 : \pi_1, \ldots, x_n : \pi_n) \to e :\sim \tau \mid \forall \overrightarrow{\delta}.\, \mathcal{C} \to (\mathcal{C}' \wedge \tau' \sim \tau)}\ \text{(T-BRANCH)}$$

図 3.10　式の型付け規則

れた制約が生成される．それらの制約が全て成り立つことがモジュール内で関数定義が妥当であるということになる．同様の妥当性をノード定義，ノードの初期値についても検査することで，モジュール全体の妥当性が検査できる．

3.4.4　使用メモリ量決定アルゴリズム

図 3.12 に使用メモリ量の決定アルゴリズムで用いる補助関数を定義する．これはサイズ変数環境 Δ の下でサイズパラメータの値を計算する関数である．

$\boxed{\mathcal{F} \Longrightarrow \mathcal{C}}$

$$\frac{\left[\begin{array}{c} \mathcal{F}(f) = (x_1 \mathtt{:} \pi_1, \ldots, x_n \mathtt{:} \pi_n) \mathtt{:} \tau \ \mathtt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}[\overrightarrow{\delta}] = e \\ x_1 \mathtt{:} \pi_1, \ldots, x_n \mathtt{:} \pi_n \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau' \mid \mathcal{C}'_f \\ \mathcal{C}_f = \forall \overrightarrow{\pi_i \downarrow}.(\bigwedge_{i \in 1 \ldots m} \mathcal{A}_i) \to (\mathcal{C}'_f \wedge \tau \sim \tau') \end{array}\right]_{f \in \mathrm{dom}(\mathcal{F})}}{\mathcal{F} \Longrightarrow \bigwedge_{f \in \mathrm{dom}(\mathcal{F})} \mathcal{C}_f} \text{(C-FUNC)}$$

$\boxed{\mathcal{N} \Longrightarrow \mathcal{C}}$

$$\frac{\mathrm{dom}(\mathcal{N}) = \{N_1, \ldots, N_n\} \qquad \left[\begin{array}{c} \mathcal{N}(N_i) = (\sigma_i, e_i) \\ \mathcal{I}, N_1 \mathtt{:} \sigma_1, \ldots, N_n \mathtt{:} \sigma_n \vdash_-^{\mathcal{T};\mathcal{F}} e : \tau_i \mid \mathcal{C}'_i \\ \mathcal{C}_i = \mathcal{C}'_i \wedge \tau_i \sim \sigma_i \end{array}\right]_{i \in 1 \ldots n}}{\mathcal{N} \Longrightarrow \bigwedge_{i \in 1 \ldots n} \mathcal{C}_i} \text{(C-NODE)}$$

$\boxed{\mathit{Init} \Longrightarrow \mathcal{C}}$

$$\frac{\left[\begin{array}{c} \mathcal{N}(N) = (\sigma, e) \\ \cdot \vdash_-^{\mathcal{T};\mathcal{F}} \mathit{Init}(N) : \tau \mid \mathcal{C}'_N \\ \mathcal{C}_N = \mathcal{C}'_N \wedge \tau \sim \sigma \end{array}\right]_{N \in \mathrm{dom}(\mathcal{N})} \qquad \left[\begin{array}{c} \cdot \vdash_-^{\mathcal{T};\mathcal{F}} \mathit{Init}(N) : \tau \mid \mathcal{C}'_N \\ \mathcal{C}_N = \mathcal{C}'_N \wedge \tau \sim \sigma \end{array}\right]_{(N,\sigma) \in \mathcal{I}}}{\mathit{Init} \Longrightarrow \bigwedge_{N \in \mathrm{dom}(\mathcal{N})} \mathcal{C}_N \wedge \bigwedge_{(N,\sigma) \in \mathcal{I}} \mathcal{C}_N} \text{(C-INIT)}$$

図 3.11　モジュール定義の妥当性検査規則

$\boxed{ev_\Delta[\![\psi]\!]}$

$$\begin{array}{lcl} ev_\Delta[\![k]\!] & = & k \\ ev_\Delta[\![\delta]\!] & = & \Delta(\delta) \\ ev_\Delta[\![\psi + \psi']\!] & = & ev_\Delta[\![\psi]\!] + ev_\Delta[\![\psi']\!] \\ ev_\Delta[\![\psi - \psi']\!] & = & ev_\Delta[\![\psi]\!] - ev_\Delta[\![\psi']\!] \\ ev_\Delta[\![\overrightarrow{\delta}]\!] & = & \sum_{\delta \in \overrightarrow{\delta}} ev_\Delta[\![\delta]\!] \end{array}$$

図 3.12　サイズパラメータの評価規則

次に図 3.13 に式の評価時に使用するメモリ量を決定するアルゴリズムを示す．このアルゴリズムは，式を走査するためのアルゴリズム $\mathcal{M}$ と `case`ブランチでのパラメータ探索のためのアルゴリズム $\overline{\mathcal{M}}$ から構成されている．アルゴリズム中では，数値の max 関数は二項演算子 $\sqcup$ によって記述されている．

$\mathcal{M}$ は型定義 $\mathcal{T}$，関数定義 $\mathcal{F}$，サイズ環境 Δ，型環境 Γ を入力，式 e を対象とし，その e の型 τ，e の評価に必要なローカル変数環境の空き容量 s，ヒープの空き容量 t，コールスタックの空き容量 u を出力する．τ にはサイズ変数を持った型が出現しうることに注意されたい．サイズ環境を伴いながら，構文木を走査することで対象の式の部分式が使用しうるリソース量を計算し，対象の式が必要とするリソース量を求める．必要とするリソース量は操作的意味論の規則から逆算される．例えば `let`式を対象とする場合，e_1 が評価できる分と x を環境に束縛した状態で e_2 が評価できる分のローカル変数環境の空き容量が必要であり，これが出力の 2 つ目 $s_1 \sqcup (1 + s_2)$ で表されている．

関数呼び出し式を対象とする場合，仮引数の具体的なサイズを補助関数 ev を用いて計算し，関数で導入されるサイズ変数に束縛し，新しいサイズ環境を作る．その新しいサイズ環境の下で関数の定義式を走査する．

$\overline{\mathcal{M}}$ では，`case`で分解する型のサイズ m が入力に追加され，式 e の代わりにブランチを対象とする．出力はこのブランチが評価される時に必要なローカル変数環境の空き容量 s，ヒープの空き容量 t，コールスタックの空き容量 u である．コンストラクタ分解の条件を満たすようなサイズパラメータ割り当

$\boxed{\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!] = (\tau, s, t, u)}$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![c^{\mathcal{B}}]\!] = (\mathcal{B}, 0, 1, 0) \qquad \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![x]\!] = (\Gamma(x), 0, 0, 0)$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![N]\!] = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![N\texttt{@last}]\!] = (\Gamma(N), 0, 0, 0)$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] = (\tau_2, s_1 \sqcup (1 + s_2), t_1 + t_2, u_1 \sqcup u_2)$
where $(\tau_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (\tau_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;(\Gamma, x:\tau_1)}[\![e_2]\!]$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = (\tau_2, s_1 \sqcup s_2 \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$
where $(\tau_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le 3)$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1\ op^{(\mathcal{B}_1,\mathcal{B}_2)\to\mathcal{B}}\ e_2]\!] = (\mathcal{B}, s_1 \sqcup (1 + s_2), 1 + t_1 + t_2, u_1 \sqcup u_2)$
where $(\mathcal{B}_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le 2)$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![f\texttt{(}e_1, \ldots, e_n\texttt{)}]\!] = (\theta\tau,\ (n + s') \sqcup \bigsqcup_{i\in 1\ldots n}(n - 1 + s_i),\ t' + \sum_{i\in 1\ldots n} t_i, (1 + u') \sqcup \bigsqcup_{i\in 1\ldots n} u_i)$
where $(\tau_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le n)$,
$\mathcal{F}(f) = \texttt{(}x_1\texttt{:}\pi_1, \ldots, x_n\texttt{:}\pi_n\texttt{)}\texttt{:}\tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}\texttt{[}\overrightarrow{\delta}\texttt{]}\ \texttt{=}\ e$,
$\theta = [\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n]$,
$(\tau', s', t', u') = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\{\pi_i\downarrow \mapsto ev_\Delta[\![\tau_i\downarrow]\!]\}_{i\in 1\ldots n};\{x_i:\pi_i\}_{i\in 1\ldots n}}[\![e]\!]$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\chi\texttt{(}e_1, \ldots, e_n\texttt{)}]\!] = (\rho^{\psi},\ 1 + \bigsqcup_{i\in 1\ldots n} s_i,\ 1 + \sum_{i\in 1\ldots n} t_i,\ \bigsqcup_{i\in 1\ldots n} u_i)$
where $(\tau_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le n),\ \chi\texttt{(}\overrightarrow{T^{\rho}}\texttt{)} \in \mathcal{T}(\rho),\ \psi = 1 + \sum_{i\in I(\chi)} \tau_i\downarrow$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{case}\ e_0\ \texttt{return}\ \tau\ \texttt{of}\ \{branch_i\}_{i\in 1\ldots n}]\!] = (\tau,\ \bigsqcup_{i\in 0\ldots n} s_i,\ t_0 + \bigsqcup_{i\in 1\ldots n} t_i,\ \bigsqcup_{i\in 0\ldots n} u_i)$
where $(\rho^{\psi}, s_0, t_0, u_0) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_0]\!],\ (s_i, t_i, u_i) = \overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{ev_\Delta[\![\psi]\!];\Delta;\Gamma}(branch_i)\ (1 \le i \le n)$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e\ \texttt{adj}\ \texttt{[}\psi\texttt{]}]\!] = (\rho^{\psi}, s, t, u) \quad$ where $(\rho^{\psi'}, s, t, u) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!]$

$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{fit}\ e_1\ \texttt{to}\ x\texttt{:}\rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3]\!] = (\tau_2, s_1 \sqcup (1 + s_2) \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$
where $(\tau_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (\tau_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma,x:\rho^k}[\![e_2]\!],\ (\tau_3, s_3, t_3, u_3) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_3]\!]$

$\boxed{\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(\chi\texttt{(}x_1 : \pi_1, \ldots, x_n : \pi_n\texttt{)} \to e) = (s, t, u)}$

$\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(\chi\texttt{(}x_1 : \pi_1, \ldots, x_n : \pi_n\texttt{)} \to e) =$
$\quad$ if $m < 1 + |I(\chi)|$ then $(0, 0, 0)$ else $(n + \bigsqcup_{a\in\mathbf{A}} s_a, \bigsqcup_{a\in\mathbf{A}} t_a, \bigsqcup_{a\in\mathbf{A}} u_a)$
$\quad$ where $\chi(T^{\rho}_1, \ldots, T^{\rho}_n) \in \mathcal{T}(\rho)$,
$\quad\quad \mathbf{A} = \{\ \{\pi_i\downarrow \mapsto p_i\}_{i\in I(\chi)}\ \mid ((\forall i \in 1\ldots n, p_i > 0) \wedge m = 1 + \sum_{i\in I(\chi)} p_i) \vee I(\chi) = \emptyset\}$,
$\quad\quad b = \{\pi_j\downarrow \mapsto k_j \mid j \in 1\ldots n, T^{\rho}_j = \rho'^{k_j}\}$,
$\quad\quad (\tau_a, s_a, t_a, u_a) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{(\Delta,a,b);(\Gamma,x_1:\pi_1,\ldots,x_n:\pi_n)}[\![e]\!]\ (a \in \mathbf{A})$

図 3.13　式の使用メモリ量決定アルゴリズム

ての集合 $\mathbf{A}$ を用いて，$\mathbf{A}$ の全て場合でブランチの式 e が評価可能なリソース量を求める．コンストラクタ分解の条件が満たされない場合，つまりそのブランチの式が実行されないと静的にわかる場合には s, t, u の全てを 0 にしている．$\overline{\mathcal{M}}$ が呼び出される，`case`式の $\mathcal{M}$ ではブランチの結果を $\sqcup$ するため，全てのリソースが 0 の場合にはその結果は無視されることになる．

このアルゴリズムはサイズパラメータの具体値を計算しながら構文木を展開するため，扱うサイズの大きさによって実行時間が変化する．また，コンストラクタのパラメータの探索も行うため，型の構造にも影響される．特に，コンストラクタでのパラメータの探索はパラメータの重複組み合わせ分の探索が必要となるため，計算量が爆発的に大きくなってしまうことがある．しかし，$\mathrm{Emfrp}^{\mathrm{BCT}}$ は小規模組込みシステム向け言語であることを加味し，扱うサイズが比較的小さく，型の構造も複雑ではないと仮定すればこのアルゴリズムでも十分に使用することができる．

ここで示したのは式の使用メモリ量決定アルゴリズムである．3.3.3 節で記述したように，モジュール全体のメモリ使用量を決定するためには，他にノード，ノードの直前値やランタイムが使用するメモリ量も考慮する必要がある．

3.4.5 健全性

これまでに定義した操作的意味論，型システム，使用メモリ量決定アルゴリズムを用いて式の型付けに対する健全性を示し，その系としてノードについても同様であることを示す．まず，使用メモリ量決定アルゴリズムの停止性を示す．型環境 Γ とサイズ環境 Δ の一貫性，およびサイズ制約のモデルを以下のように定義する．

定義 3 (Consistency between type environment and size environment)**.**
$\bigcup\{\mathrm{FV}(\Gamma(x)\downarrow) \mid x \in \mathrm{dom}(\Gamma)\} \subseteq \mathrm{dom}(\Delta)$ が成り立つ時，「型環境 Γ とサイズ環境 Δ に一貫性がある」と言い，(Γ, Δ)-consistent と書く．

定義 4 (Model of size constraint)**.**
$\mathrm{FV}(\mathcal{C}) \subseteq \mathrm{dom}(\Delta)$ であるようなサイズ環境 Δ とサイズ制約 $\mathcal{C}$ について，$\mathcal{C}$ 内の自由変数の出現を全て Δ によって置換して得られた制約が妥当である時，「Δ は $\mathcal{C}$ のモデルである」と言う．

これらの定義を用いて式の使用メモリ量決定アルゴリズムについて定理 5 を示す．

定理 5 (Termination of the memory usage estimation algorithm)**.**
仮定

- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C}$.
- (Γ, Δ)-consistent.
- 式 e に現れる全ての関数 g は，$f \geq_{\mathcal{F}}^{*} g$ を満たす
- Δ は $\mathcal{C}$ のモデルである

のもとで，

- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!]$ は有限ステップで停止する
- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s, t, u)$ のとき，$\tau_{\mathcal{M}} = \tau$

が成り立つ．

Proof Sketch. 三つ組 $(f, ev_{\Delta}[\![\overrightarrow{\delta}]\!], e)$ の辞書式順序についての帰納法による．ただし f は関数名，$\overrightarrow{\delta}$ は f の尺度関数，e は式である．詳細は付録 A.1 を参照されたい． □

$\boxed{H, \mathcal{T} \vdash l : \sigma}$

$$\frac{H(l) = c^{\mathcal{B}}}{H, \mathcal{T} \vdash l : \mathcal{B}}\ \text{(H-CONST)} \qquad \frac{\begin{array}{l} H(l) = \chi[k](l_1, \ldots, l_n) \\ \chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \\ k = 1 + \sum_{i \in I(\chi)} k_i \end{array} \quad \left[\begin{cases} H, \mathcal{T} \vdash l_i : \rho^{k_i} & (i \in I(\chi)) \\ H, \mathcal{T} \vdash l_i : \sigma_i \quad \wedge \quad \sigma_i \downarrow \leq T_i^\rho \downarrow & (\text{otherwise}) \end{cases} \right]_{i \in 1 \ldots n}}{H, \mathcal{T} \vdash l : \rho^k}\ \text{(H-CTOR)}$$

図 3.14　ヒープ上の値の型付け規則

ノード更新式について，対象のノードが依存するノード全てを入力型環境として設定することで，以下の系 1 が成り立つ．

系 1 (Termination of the memory usage estimation algorithm for node update expression)**.**
すべてのノード N について，$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{;\Gamma_\mathcal{N}}[\![\mathcal{N}(N)]\!]$ は有限ステップで停止する．ただし $\Gamma_\mathcal{N} = \mathcal{I}, N_1 : \sigma_1, \ldots, N_n : \sigma_n$ である．

以上により，使用メモリ量決定アルゴリズムが有限ステップで停止することが示された．以降では暗黙に使用メモリ量決定アルゴリズムの停止性を利用する．

次に，ヒープ上の値が表す型と各環境の一貫性を定義する。図 3.14 にヒープ H 上の位置 l についての型付け規則を示す．これはヒープ H 上で l からたどって得られる値がどの型に属しているかを表現する．ヒープと各環境の一貫性の定義は次の通りである．

定義 6 (Consistency between node location environment, local variable environment, heap, type environment, and size environment)**.**

- (Γ, Δ)-consistent
- すべての $x \in \mathrm{dom}(E)$ について $H, \mathcal{T} \vdash E(x) : \sigma$ かつ $\sigma \downarrow \leq ev_\Delta[\![\Gamma(x) \downarrow]\!]$
- すべての $n \in \mathrm{dom}(\mathcal{L})$ について $H, \mathcal{T} \vdash L(n) : \sigma'$ かつ $\sigma' \downarrow \leq ev_\Delta[\![\Gamma(\mathrm{node\_name}(n)) \downarrow]\!]$

が全て成り立つ時，「ノード位置環境 $\mathcal{L}$，ローカル変数環境 E，ヒープ H，型環境 Γ，サイズ環境 Δ に一貫性がある」と言い，$(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent と書く．ただし $\mathrm{node\_name}(n)$ は n のノード名を表す補助関数[*3]である．

最後に，定理 7 に式の型付けに対する健全性を示す．

定理 7 (Soundness of typing of expression)**.**
仮定

- $\Gamma \vdash^{\mathcal{T};\mathcal{F}}_f e : \tau \mid \mathcal{C}$.
- $(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent.
- 式 e に現れる全ての関数 g は，$f \geq^*_\mathcal{F} g$ を満たす
- Δ は $\mathcal{C}$ のモデルである
- $\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!] = (\tau_\mathcal{M}, s_\mathcal{M}, t_\mathcal{M}, u_\mathcal{M})$.

のもとで，すべての $s \geq s_\mathcal{M}, t \geq t_\mathcal{M}, u \geq u_\mathcal{M}$ について，ある l, t', H', σ が存在し，

- $[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_\mathcal{L} e \Downarrow_u l; [t']H'$

[*3] n がノード名であるときはそのままの名前を返し，n が直前値であるときは対象の名前部分を返す．

- $t' \geq t - t_{\mathcal{M}}$
- $H', \mathcal{T} \vdash l : \sigma$
- $\sigma \downarrow \leq ev_{\Delta}[\![\tau \downarrow]\!]$
- σ と τ のそれぞれのサイズパラメータを除いた型名が一致する

を満たす.

Proof Sketch. 定理 5 の証明でも使用した，三つ組 $(f, ev_{\Delta}[\![\overrightarrow{\delta}]\!], e)$ の辞書式順序についての帰納法による．詳細は付録 A.2 を参照されたい. □

ノード更新のはじめにはローカル変数は存在しないこと，ノード定義式のトップレベルにはサイズ変数が存在しないため $\Delta, \mathcal{C}$ は空であることに注意すると，ノードの型付けについての系 2 が得られる.

系 2 (Soundness of typing of node)**.**
仮定

- $\Gamma_{\mathcal{N}}(N) = \tau_N$.
- $(\mathcal{L}, \cdot, H, \Gamma_{\mathcal{N}}, \cdot)$-consistent.
- $\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\cdot;\Gamma_{\mathcal{N}}}[\![\mathcal{N}(N)]\!] = (\tau_{\mathcal{M}}, s_{\mathcal{M}}, t_{\mathcal{M}}, u_{\mathcal{M}})$.

のもとで，ある l, t', H', σ が存在し，

- $[s_{\mathcal{M}}]\cdot \mid [t_{\mathcal{M}}]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \mathcal{N}(N) \Downarrow_{u_{\mathcal{M}}} l; [t']H'$
- $H', \mathcal{T} \vdash l : \sigma$
- $\sigma \downarrow \leq \tau_N \downarrow$
- σ と τ_N のそれぞれのサイズパラメータを除いた型名が一致する

を満たす.

以上に示した定理，系から Emfrp$^{\mathrm{BCT}}$ の操作的意味論，型システム，使用メモリ量決定アルゴリズムには健全性が成り立つ．つまり，正しく型付けされ，サイズ制約が妥当 (充足可能) であるノードについて，使用メモリ量決定アルゴリズムによって得られたリソース量を操作的意味論の入力リソース量とした時，リソースが不足することなくノード更新計算が行われる.

3.5 評価

Emfrp$^{\mathrm{BCT}}$ の処理系を実装した．実装に伴う注意点を説明したのち，各種オーバーヘッドを計測し議論する.

3.5.1 Emfrp$^{\mathrm{BCT}}$ コンパイラ

実装した処理系では，Emfrp$^{\mathrm{BCT}}$ コードは C 言語コードへとコンパイルされる．処理系によって出力された C コードをコンパイルすることで実行バイナリを得る．コンパイルされたプログラムは，3.4.2 節で示した操作的意味論を忠実に再現するため非効率に振る舞う.[*4]例えば，規則 E-CALL や規則 E-CTOR の実装の際には，C コンパイラによる意図しない最適化を防ぐために，ローカル変数確保やスタックポインタ操作などを独自に管理している．外部からの入力や外部への出力の形式やユーザー

[*4] C 言語では，ローカル変数のための領域はレジスタに割り当てられるかスタックポインタを操作することでスタックメモリ上に確保される．C コンパイラは最適化のために，関数先頭にてローカル変数のための領域を一括で確保することがある．この振る舞いは第 3.4.2 節にて定義した変数定義の意味論と合致しない.

表 3.1　DupCheckモジュールにおける Listのサイズとコンパイル時間の比較

N	Type check[sec]	Memory estimate[sec]	Total[sec]
4	0.015466	0.000049	0.015783
10	0.015334	0.000068	0.015680
20	0.015473	0.000128	0.015876
50	0.015486	0.000233	0.015993
100	0.015504	0.000530	0.016371
200	0.015121	0.000830	0.016227
300	0.015402	0.001312	0.016992
400	0.014178	0.001841	0.016312
500	0.015623	0.005209	0.021105

が追記する必要のあるコードについては Emfrp を踏襲している (第 2.4 節を参照).

BCT は Emfrp でのヴァリアント型に実行時サイズ情報が追加されたデータとしてコンパイルされる．言語ランタイムには BCT を管理するために，Emfrp でのヴァリアント型のためのメモリ管理を拡張したある種のマークスイープ GC が含まれる．実装に使用した実行時使用メモリ決定アルゴリズムでは，GC で使用されるメモリ領域についても加味されている．

基本型の値の実行時表現について，形式的定義と実装に差異がある．形式的定義では，基本型の値は BCT の値と同様にヒープ上に配置され，変数スタックからのポインタを経由して参照していた．しかし，実装では変数スタックに値を直接割り当て，代入の際には値をコピーして使用するようなコードが出力される．

また，使用メモリ量決定アルゴリズムについて，関数呼び出し式の解析において同じサイズパラメータによる複数回の走査が行われないようにするなど，高速化を期待した工夫がなされている．

3.5.2　実験設定

実装した処理系を用いて評価実験を行った．処理系の実装は OCaml (version 4.11.1) を使用し，外部ライブラリとして SMT ソルバ Z3 (version 4.8.9.0) [13] を使用した．出力された C コードは clang (version 12.0.0) によってコンパイルした．コンパイルに際し小規模組込みシステムを対象としているため，プログラム容量に関する最適化のためのオプション-Os を設定した．測定はいずれも MacBook Air (CPU: Intel Core i7 2.2 GHz, Memory: 8 GB, OS: macOS Catalina 10.15.6) 上で行った．

3.5.3　コンパイル時間

3.4.4 節で提案した使用メモリ量決定アルゴリズムはプログラム中で使用されている型のサイズや，型の内部で使用されている再帰型の個数に起因して実行時間が変化する．3.3.4 節で提示した 2 つのモジュール DupCheckモジュールと Top10Sumモジュールについて，履歴ノードのサイズを変化させて，プログラムのコンパイルにかかる時間を計測した．計測の結果を表 3.1 と表 3.2 に示す．それぞれのプログラムでは List[$N+1$]型と Heap[$2N+1$]型を使用している．

DupCheckモジュールでは，使用メモリ量決定にかかる時間が緩やかに変化している．List型は case式による分解の際，探索すべきサイズが一意に定まり高々リストの長さ分の関数展開が行われ，構文木が走査される．そのためサイズの増加によるアルゴリズムの計算時間の増加は緩やかであることが予想され、実際の測定でもその傾向が確認された．

表 3.2　Top10Sumモジュールにおける Heapのサイズとコンパイル時間の比較

N	Type check[sec]	Memory estimate[sec]	Total[sec]
10	0.018962	0.013779	0.033102
20	0.018160	0.192269	0.210782
30	0.019086	1.028855	1.048550
40	0.021169	3.467531	3.489383
50	0.016734	8.922994	8.940415
60	0.019698	19.372921	19.393433
70	0.028259	36.680560	36.710777
80	0.017129	64.181296	64.200058
90	0.017927	104.388627	104.408470
100	0.019529	163.357405	163.379870

表 3.3　DupCheckモジュール ($N = 4$) における実行時間とプログラムサイズの比較

$N = 4$	Exec[sec]	text[byte]	data[byte]	bss[byte]
Emfrp	0.358	1374	56	240
Tuple	0.840	2419	48	384
List	1.702	2395	56	640

表 3.4　DupCheckモジュール ($N = 30$) における実行時間とプログラムサイズの比較

$N = 30$	Exec[sec]	text[byte]	data[byte]	bss[byte]
Emfrp	1.467	6312	64	1280
Tuple	5.045	8149	56	1840
List	12.063	2417	56	3552

表 3.5　Top10Sumモジュール ($N = 10$) における実行時間とプログラムサイズの比較

$N = 10$	Exec[sec]	text[byte]	data[byte]	bss[byte]
Emfrp	0.484	3397	56	472
Insertion	2.339	5547	48	720
Heap tree	2.090	3582	56	3024

一方で，Top10Sumモジュールでは，比較的小さいサイズでも使用メモリ量の決定に時間がかかってしまう．これは Heap型には Heap型が 2 つ含むコンストラクタがあることに起因する．case式で木を分解する際にサイズの条件を満たすようにサイズ変数の割り当てと探索が網羅的に行われるため，計算時間が爆発的に増えてしまうことが予想される．実際の測定からもその傾向が確認できた．

どちらのプログラムでも N によらず型検査にかかる時間はそれぞれほぼ一定である．型検査中に構築される制約の形は型のサイズによらないので，一定の時間で充足判定ができているということが確認できた．

表 3.6　Top10Sumモジュール ($N = 30$) における実行時間とプログラムサイズの比較

$N = 30$	Exec[sec]	text[byte]	data[byte]	bss[byte]
Emfrp	1.409	11585	56	1272
Insertion	6.692	22976	56	1840
Heap tree	5.533	3573	56	8784

3.5.4　実行時間とプログラムサイズ

BCT を使用した事によるプログラムサイズへの影響と実行時のオーバーヘッドを調べるための計測を行った．DupCheckモジュールと Top10Sumモジュールについて，第 3.2 章で示した Emfrp コードを既存の Emfrp コンパイラ[*5]で変換したもの，タプルを用いて $\text{Emfrp}^{\text{BCT}}$ で再現実装したもの，BCT を用いて記述したものの 3 つを対象として，ランダムな入力を用いて 10^7 回のイテレーションにかかった時間とプログラムサイズ (text，data，bss) を計測した．

DupCheckモジュールでは，履歴ノード historyの型が List[5]($N = 4$) と List[31]($N = 30$) であるものを測定した．測定の結果をそれぞれ表 3.3 と表 3.4 に示す．どちらの場合についても，リストを使用したプログラムよりタプルによる実装の方が高速に処理が行われている．プログラムコードが格納される text 領域について，$N = 4$ の場合には大きな差はない．しかし $N = 30$ の場合にはタプルの方では容量が多くなっている．これはコンストラクタ適用がインライン化されることや，要素の重複を探すための処理が if式の羅列として実装されていることに起因する．一方でリストを使用した方では，履歴数が変化しても text 領域の大きさはほとんど変わりがないが，bss 領域が有意に増加している．これは管理する履歴数が増えたことで関数呼び出しの回数が増え，使用するスタックの容量が増加するためである．プログラムサイズ全体としてはリストを用いた例の方が省メモリである．

Top10Sumモジュールでは，履歴ノード hの型が Heap[21]($N = 10$) と Heap[61]($N = 30$) であるものを測定した．測定の結果を表 3.5 と表 3.6 に示す．実行時間について，要素数が変わってもヒープ木を用いて実装した例の方が高速に動作している．一方でプログラムサイズについては，DupCheckモジュールの例と同様の傾向が見られる．$N = 30$ の時のタプルの挿入ソートによる実装では text 領域が肥大化している．これはタプルの適切な位置に値を挿入するために全ての要素番号に対してコンストラクタ適用を記述しているためである．

DupCheckモジュールと Top10Sumモジュールのどちらの測定結果にも共通することとして，既存の Emfrp によるプログラムは高速かつ比較的省メモリに動作している点が挙げられる．既存の Emfrp では，Emfrp 上の変数や関数呼び出しを C 言語での変数や関数呼び出しに対応させた変換を行う．そのため，ローカル変数をレジスタに割り当てるなどの最適化がされやすく，計算が高速に行われる．一方で，測定のために実装した $\text{Emfrp}^{\text{BCT}}$ 処理系は，意味論と挙動を合わせるためにスタックを自前で管理するなど計算を行う上でオーバーヘッドがかかる設計になっている．また，既存の Emfrp 処理系ではノードの生存情報を解析することで，ノードが占めるメモリの解放を早期に行う仕組みがあるため，より省メモリに計算を行うことができている．

3.5.5　イテレーション内の実行時間

プログラムの実行時間について，イテレーション中の段階ごとに計測を行った結果を表 3.7 と表 3.8 に示す．これは $N = 30$ の場合について，それぞれのプログラムの実装方針ごとに，ノード更新処理に

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

表 3.7　DupCheckモジュール ($N = 30$) におけるイテレーション処理にかかる時間

$N = 30$	update[sec]	mark[sec]	refresh[sec]	total[sec]
Tuple	14.174	5.293	5.980	45.962
List	19.612	7.087	5.959	53.227

表 3.8　Top10Sumモジュール ($N = 30$) におけるイテレーション処理にかかる時間

$N = 30$	update[sec]	mark[sec]	refresh[sec]	total[sec]
Insertion	15.850	5.300	5.975	47.651
Heap tree	11.376	7.743	6.600	46.210

かかった時間 (update)，ガベージコレクションのためのマークフェーズにかかった時間 (mark)，メモリのリフレッシュ (スイープフェーズ) にかかった時間 (refresh)，プログラム全体の実行時間 (total) を累積して計測したものである．時間の単位はそれぞれ秒である．時間計測はイテレーションループ内に計測用のコード (clock関数を使用) を挿入して行った．計測コードによるオーバーヘッドが含まれているため，表 3.4 と表 3.6 の結果よりも全体時間が増加している．

DupCheckモジュールに関して，ヒープ領域に確保される履歴ノードはリストの場合とタプルの場合でほぼ同じメモリ容量のため，リフレッシュにかかる時間はほぼ同程度だった．一方で，リストをマークするためにオーバーヘッドがあることが計測できた．さらにノードの更新時間について，リストを使用した場合の方が実行に時間がかかっていた．使用したプログラムを比較すると，タプルによる実装の方ではパターンマッチは履歴を更新するために 1 回のみ必要だが，リストによる実装では，履歴の長さと比例した回数のパターンマッチが行われる．そのため，リストによる実装では関数呼び出しやパターンマッチの増加により，タプルによる実装と比べノードの更新時間が増加してしまうことが測定から確認できた．

Top10Sumモジュールに関して，ヒープ領域に確保されるオブジェクトは，タプルによる挿入ソート実装の場合には N 個であるが，ヒープ木による実装では $2N + 1$ 個である．そのため，ヒープ木による実装では，挿入ソートによる実装と比べヒープ上のオブジェクトに対するマークやリフレッシュの時間が増加している．一方で，それぞれの実装を比較すると，挿入ソート実装では履歴長に比例したパターンマッチが必要なのに対し，ヒープ木実装では木の高さに比例した回数のパターンマッチが行われる．多くの場合には木の高さの方が履歴長よりも小さいため，ヒープ木による実装の方がノードの更新時間は短くなっている．

3.5.6　考察

評価実験から，BCT を使用したプログラムでは処理の繰り返しが再帰関数で表現されることでコードサイズの肥大化を防ぐこと，BCT のオブジェクトにマークに多少のオーバーヘッドがかかることが確認できた．ノードの更新時間に関しては，それぞれの場合で計算方法 (アルゴリズム) が異なっているため単純な比較を行うことができない．DupCheckモジュールの場合には，リストによる実装は履歴数の変更に強いプログラムとなっている．しかし，要素へのアクセスは再帰関数とパターンマッチの組み合わせによって行うため，タプル実装と比べて時間のかかる処理になってしまっている．一方で，Top10Sumモジュールの場合には，既存の Emfrp では表現することの難しかったヒープ木を使用したことで，より時間効率のよい処理を記述することができた．リアクティブシステムの応答性を向上させるためにも，計算効率の良いデータ構造を用いてプログラムの高速化を図ることは有用である．

タプルを利用して愚直なデータ管理を行う場合と比べ，BCT を使用したプログラムでは再帰関数呼び出しとパターンマッチの多用が想定される．そのため，処理系の性能向上のために関数呼び出しやパターンマッチの処理を高速に行うように改善することが今後の課題となる．また，サイズパラメータの情報を用いることで静的に再帰関数呼び出しの上限回数がわかるため，インライン展開を行うことで処理の高速化が期待できる．しかし，関数のインライン展開はプログラムコードが肥大化する可能性があるため，動作環境ごとに最適化するかどうか判断すべきである．

3.6 関連研究

3.6.1 Sized Types

本研究の提案手法である Bounded-Construction-Types は Sized Types [34] に強い影響を受けている．Sized Types では BCT と同様のオブジェクトサイズの最大値に加え，最小値を表す値を型の中に含む．Sized Types による型システムを用いることで，ストリームが値を必ず生成すること (productivity) などの性質を保証し，組込みシステム上で安全に関数型言語を使用することができる．

Sized Types の体系と本研究の提案手法では，`fit`式によって実行時のサイズ情報を扱うことができる点が最も大きな差異となっている．Emfrp では`@last`により自身の直前値を参照し，処理を行うことが多い．ノードの型が BCT である場合には，現在値と直前値が同じ型を持っているため，ノード定義の際にうまく型を合わせる必要がある．Emfrp$^{\mathrm{BCT}}$ では，`fit`式によって実行時のサイズ情報を利用してサイズを減らす方向にサイズをキャストできるため，要素の追加などの処理が簡潔に記述できる．一方で Sized Types の体系では，そのようなキャストが行えない場合にはあらかじめダミーのデータを要素として持ち，常にサイズが同じであるようにプログラミングする必要があるため，データの扱いが不自然になってしまう．

以下に示す `Top10Sum`モジュール内の `h`ノードを簡略化したコード片を具体例として説明する．

```
1 node h : Heap[21] init (E adj[21]) =
2   fit h@last to
3   | hl:Tree[19] -> insert(input, hl)
4   | fail        -> insert(input, delMin(h@last))
```

Emfrp に対し Sized Types を単純に導入した場合，つまりノード `h`を `fit`式を使用せずに記述する場合いくつかの記述方法が考えられる．1 つめは以下に示すようにあらかじめダミーのデータ構造を設定することで，`delMin`により常にサイズを減らす方法である．

```
1 node h : Heap[21] init (DUMMY_HEAP) =
2   insert(input, delMin(h@last))
```

この方法では，`DUMMY_HEAP`を正しい構造とサイズで設定する必要があり，場合によっては初期値として与えることが難しい．また，用途によっては `Int`型のダミーデータを用意することが適切でない場合もあるため，例えば `Int`のオプショナル型を利用する場合もあり，`delMin`が不必要に複雑になりうる．

2 つめは，実行時のヒープ木の大きさを計算する関数 `size`を用いて `if`式で処理を分岐する方法である．これを以下に示す．

```
1 node h : Tree[21] init (E adj[21]) =
2   if size(t@last) <= 19 then
3     insert(input, CAST_SIZE(h@last))
4   else insert(input, delMin(h@last))
```

ヒープ木の容量に余裕がある場合には `h@last`に対して `insert`関数を呼び出すために，引数の型 (サイズ) を合わせる必要がある．このコード片では `CAST_SIZE`関数が構造を保ったままサイズパラメータを減少させる操作を表しているが，既存の Sized Types でこのような処理を関数として記述することは

できない．式としては，必要なサイズを表せるまで `case`式による分解を行うことで実現可能である．BCT の値を `case`式で分解することで，サイズパラメータを減少させることが可能であるため，例えばリストのような単純なデータ構造に対しては，`case`式を必要な回数分ネストさせることで，必要なサイズパラメータを持つリストを得ることができる．しかし，ノード `h`のように木構造を分解し所望のサイズを得る場合，単純な `case`式のネストではなく木構造の取りうる形すべてを考慮した複雑な式が要求される．マクロなどにより式の生成を行うことは可能だが，`case`式によって与えられる大量の分岐によって，所望のサイズを得た場合と得られなかった場合のコードが複製され，プログラムが肥大化してしまう．小規模組込みシステムを対象とした本研究ではコード複製によるプログラムの肥大化は避けるべきである．

以上の理由により，Emfrp の`@last`を使用するプログラミングスタイルに合わせるために，`fit`式の導入は有用である．そして `fit`式で使用するために，BCT の実行時表現には実際のサイズ情報が保持されている．

Sized Types とリージョンによるメモリ管理 [71] を組み合わせ実行時に使用するリソース量を決定する手法が Hughes らによって提案されている [33]．リージョンを確保する際にそのサイズ情報を付与し，型付けの際のエフェクトとしてスタックやヒープへのストアの個数を持つことで，型検査に合格したプログラムが一定のメモリ領域で計算可能であることを示している．彼らの言語では，オブジェクトをヒープに確保する際に必ずどのリージョンに確保されるのかを明示するので，本研究のように `case`式の部分でリソース使用の上限を得るための探索を行わなくてもよい．ただし，リージョンの適切な大きさをユーザーが事前に計算し指定する必要がある．

3.6.2 配列によるデータ構造の表現

実行時に要素数が変化するデータ構造は，手続き型言語においては配列を用いて表現されることも多い．木などのデータ構造を配列の添字番号をポインタのように扱うことで表現することができる．しかし，このような使い方をメモリ効率よく行うためには配列が破壊的に変更可能である必要がある．Emfrp を拡張する際に，純粋な FRP 言語であることを保存するため配列の導入は行わなかった．仮に配列を導入する場合，ノード更新の停止性を保証するためにループ回数を定数回に限るなどの使用しづらい制限が予想されたため，本研究ではサイズ情報を持った再帰データ型と原始帰納的関数を導入するに至った．

純粋な関数型言語上でメモリ効率よく配列を扱う試みとしては，GPU 向けの関数型言語 Futhark [18, 28] が挙げられる．Futhark では配列を線形型によって型付けすることでインプレースなアップデートが可能である．この性質を行列に対して応用することで，関数型言語でも効率よい GPU 向けのプログラムを記述できる．最近では行列のサイズ情報を伴った関数の型を記述するための Size Type と呼ばれる機構 [26, 27] が導入され，一部のオブジェクトサイズを静的に判断できる．本研究の BCT とは目的が異なるが，類似した記法が用いられている．

3.6.3 FRP 言語，データフロー言語，その他の言語

Hume [22] は実時間組込みシステムを対象とした DSL であり，Box と呼ばれるイベント駆動型の状態機械とそれらの協調動作の記述によってプログラムを構成する．Hume はサポートする言語機能ごとに複数のレベルが設定されており，Emfrp は FSM-Hume と呼ばれるレベルに対応している．本研究で提案した $\text{Emfrp}^{\text{BCT}}$ は PR-Hume のレベルに対応し，原始帰納的関数や帰納的なデータ構造がサポートされている．FSM-Hume や PR-Hume にはコストモデルが定義され [23]，$\text{Emfrp}^{\text{BCT}}$ と同様にリソース領域の残り容量や残り時間を伴った操作的意味論が用いられる [21, 73]．リソース量の解析には Sized Types を用いたものや後述する AARA を用いたものが提案されている [36]．一方，本研究の

$\text{Emfrp}^{\text{BCT}}$ と異なり，Hume は言語上でデータ構造の大きさを持っていないため，上限以内かを確かめる `fit`式に相当する表現を直接行うことができず，別の形，例えばデータ構造の大きさを測る関数を用いて実現する必要がある．Hume は多くの静的解析を組み合わせることでリソース量などの解析を行っているが，用いた関数が確かに「データ構造の大きさについて表している」ことを静的解析によって正確に得ることは難しく，またそれぞれの分岐に対する解析では大きさに関する条件を正確に引き継いで行うことも難しい．

マイクロコントローラ (ATmega328) を用いたマイコンボードである Arduino 向けに設計された小規模組込みシステム向け FRP 言語として Juniper [25] がある．Juniper が提供する C++ のテンプレートに相当する機構では，通常の型パラメタに加えて capacity variable と呼ばれるパラメタを指定することができる [24]．このパラメタを用いることで，配列を含むレコード中の配列のサイズを指定すること等が可能である．ただし，C++ のテンプレートへの変換として実現されており，本研究での提案手法のように，再帰的なデータ型やそれを扱う再帰的な関数について最大データサイズを保証することはできない．また，Juniper では再帰的なデータ型を定義することも可能であるが，それについてはデータサイズを静的に規定する機構は提供していない．

本研究とは別のアプローチによるリソース管理を行っている FRP 言語として，Krishnaswami らの提案した FRP 言語 [38] が挙げられる．彼らの提案する言語では，プログラムの開始前にあらかじめリソースを表す特殊な値を用意し，その値を使い回すことで一定のメモリ量で計算を行う．プログラマが適切にリソースの管理を行うことで，リスト構造を扱うことができる．加えて，線形型付けされたリソース値によって高階関数や高階の時変値を space leak することなく扱うことができ，FRP として表現力に富んだ記述が可能である．

Lustre [20, 45] はリアルタイムリアクティブシステム向けの同期的データフロー言語である．時変値を組み合わせて処理を記述する点や，1 ステップ前の値を取得できる機能 (Emfrp では`@last`，Lustre では `pre`) が用意されている点など，Emfrp との共通点の多い言語である．Emfrp と同様に再帰呼び出しなどのループや再帰データ型の使用を禁止することで最悪計算時間やメモリ使用量を見積もることができる．Lustre では複数個のデータに対し同様の処理を記述するために配列とそれを操作するための専用の命令群 (`map`，`reduce`など) が用意されている．Lustre では `DupCheck`の例ではそれらの命令を用いても簡潔な記述を行うことができるが，木構造を扱うようなプログラムの記述には向いていない．一方で $\text{Emfrp}^{\text{BCT}}$ では BCT を用いてリスト型を定義することで，Lustre での配列と同等の機能を使用することができる．

3.6.4 その他のリソース量推定手法

依存型や Indexed Types [81] によってオブジェクトのサイズ情報を型に含めることができる．Indexed Types は代数的データ型に対してサイズの注釈や制約が記述できる型システムである．型定義の際に柔軟な制約が記述できることが Sized Types や本研究との違いであるが，本研究では，サイズに関する整合性よりも，実行時に必要なリソース量に注目しているため，型が実行時にどれだけリソースを占めるのかをユーザーに記述させることにしている．依存型システムは値に依存した型の記述を許す型システムで，述語論理と対応していることが知られている．依存型を用いることで，サイズ情報を持った型を定義，使用することができる．しかし，依存型ではプログラムの記述と同時にサイズに関する証明も必要になっていまい，ユーザーにとって負担である．本研究ではサイズの情報のみをユーザーに記述させ，整合性の検査はソルバを用いて自動で行うため，ユーザーの負担は比較的少ない．また，ユーザーの負担を軽減するために，依存型などによって得られた制約を SMT ソルバなどで自動的に解くことが考えられるが，制約によっては求解が決定不能になってしまうことが考えられる．今回のユースケースは常に充足可能な制約が入力されるとは限らない型 (サイズ) 検査である．正しい制約が必ず

しも求解できない場合，ソルバに入力した制約がそもそも充足不可能なものなのか，ソルバの求解に時間がかかっているのかの区別を行うことができない．制約の解がもとまらなかった場合にその原因がわからなければユーザーに対するエラーメッセージを適切に報告することはできない．本研究では解くべき制約がプレスバーガー算術の範疇であるため，どのような場合にも型検査が決定可能であるということが優位な点として挙げられる．型検査が決定可能であるため，制約の解が求まらなかった場合 (型検査に失敗した場合) には入力した制約が充足不能だったと判断することができ，適切なエラーを報告できる．

Automatic Amortized Resource Analysis (AARA) [30] は Hofmann らによって導入された，静的にプログラムの使用リソースを解析する手法である．AARA では，本研究や Sized Types のようなオブジェクトサイズの情報ではなく，ポテンシャルと呼ばれる数値を付与した型システムと線形計画ソルバによって静的に使用リソース量を推定する．近年では，Hoffmann らが OCaml に対してこの手法を適用している [29]．AARA ではポテンシャルを型に付与するためユーザーが直感的にリソース量を設定することが難しい，そのため本研究ではデータ型の直接のサイズ情報を型に付与するアプローチをとった．本研究では，使用リソース量の決定のために構文木をサイズや型の構造に依存して網羅的に走査するため，解析に時間がかかってしまう場合があるが，AARA では型付けによって得られた制約から線形計画によって使用リソースを推定するという異なったアプローチであるため，この手法を取り入れることで，使用メモリ量決定アルゴリズムの高速化が期待される．

3.7 むすび

本章では，小規模組込みシステム向け FRP 言語 Emfrp に対し，「構築できる構造の最大サイズ」を伴った再帰データ型 Bounded-Construction-Types (BCT) を導入した言語 $\text{Emfrp}^{\text{BCT}}$ を提案した．再帰データ型の導入により，既存の Emfrp では冗長な記述を要求されていたプログラムの簡潔な記述や，データ管理の効率的な扱いが可能になった．$\text{Emfrp}^{\text{BCT}}$ は再帰データ型が導入されながらも，Emfrp の持つノード更新の停止性と使用メモリ量を静的に決定できる性質を保存していることを $\text{Emfrp}^{\text{BCT}}$ の操作的意味論，型システム．使用メモリ決定アルゴリズムを形式化し，その健全性を示すことで証明した．さらに，$\text{Emfrp}^{\text{BCT}}$ の処理系を実装し，コンパイル時間や実行時間のオーバーヘッドを計測した．

今後の展望としては以下に挙げる発展が考えられる．

1. 型およびサイズに対する多相性の導入
2. 相互再帰への対応
3. サイズの和以外にも対応する尺度関数の導入

現状のメモリ使用量決定アルゴリズムは，サイズ情報を伴って構文木を走査するため，サイズが大きい時や複雑な BCT を扱う際には探索時間がかかる．そこで，本研究で提案するアルゴリズムを本番環境用にコンパイルするためのアルゴリズムとして，リソースの十分にある環境でテストをする際に使用できるような高速でより過大に近似するアルゴリズムの開発が求められる．

第 4 章

CPU リソース：$\mathrm{Emfrp}^{\mathrm{HT}}$

4.1 まえがき

本章の主な貢献は **Heavy-task(第 4.2.1 節参照) を扱う機構を備えた FRP 言語 $\mathrm{Emfrp}^{\mathrm{HT}}$ を設計し，その実行モデルを実現するために push 型の FRP 言語処理系に対する非同期タスク実行基盤による拡張を提案したこと**である．具体的には以下である．

- 現実的な問題 (ロボット競技マイクロマウス) を例題とし，リアクティブな振る舞い (Emfrp などの外部 DSL による FRP 記述) と Heavy-task の協調実行における問題点を分析した．
- Emfrp 言語を基礎として，リアクティブな記述中で Heavy-task を自然に扱うための機構を備えた拡張言語 $\mathrm{Emfrp}^{\mathrm{HT}}$ を提示した．拡張された言語機構には，タスクと共有リソースの定義機構，Heavy-task のための時変値定義機構が含まれる．
- 前述の言語機構を実現するための言語ランタイム設計および実装方法を提案した．言語ランタイムには，Heavy-task 実行のための非同期タスク実行基盤 (シンプルなスケジューラとリソース制御機構) が含まれる．
- 実機による評価実験を行い，提案した言語機構によって応答性の悪化を低減できることの確認および言語機構の導入に伴うオーバーヘッドを計測した．

FRP では，それぞれの時変値の更新処理は無視できるほど小さいと仮定されることが多い．しかしながら，実行環境が小規模な組み込みシステムにおいても，例えばグラフアルゴリズムなどの比較的時間のかかる処理を行うことがあり，その仮定が成り立たないことがある．本論文では，そのような**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**を **Heavy-task**(ヘビータスク) と呼ぶ (詳細な説明は第 4.2.1 節)．Emfrp によって記述されたシステムの時変値更新処理中に Heavy-task が存在すると，入力に対する反応時間が増加し応答性が悪化する．これは，リアクティブな処理の実行中に，Heavy-task が CPU リソース (処理時間) を占有してしまうことで引き起こされる問題で，システム全体の応答性低下を招いてしまう．この問題は **Reactive Thread Hijacking Problem (RTHP)** [72] として知られ，解決する手法として Actor-Reactor モデル [72] が提案されている．本章で提案する $\mathrm{Emfrp}^{\mathrm{HT}}$ は RTHP に対する別の解決策を提示する言語である．

$\mathrm{Emfrp}^{\mathrm{HT}}$ に導入された Heavy-task を扱うための言語機構は，CPU リソースの適切な分割をメモリリソースの競合なく行うことを図るものである．これを利用することで，RTHP によって引き起こされる CPU リソースの占有を防ぎ，システムの不安定な振る舞いを避けながら，安定なシステムを安全に記述することが可能である．

本章の内容は文献 [78] にて発表済みである．加えて本章では，実機上での評価実験を行い提案した言語機構の有用性を確かめた．

4.2 動機：実行に時間のかかる処理

ロボット競技「マイクロマウス」による例を通して，リアクティブな振る舞いと Heavy-task 実行の協調について分析し，言語拡張の必要性について論じる．

4.2.1 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複雑なデータ構造を用いた処理が要求されることは多い．そのようなデータ構造の利用は，メモリの使用量や計算時間を増大させやすい．また，数値微分のように反復を伴う計算や行列演算，画像処理など組込みシステムにおいても有用なアルゴリズムが比較的処理時間を要求することは珍しくない．Emfrp はノード (時変値) 更新を依存順序に従ってシングルスレッド環境で順番に実行する．ノード更新にて時間のかかる計算処理が要求されると，その処理が CPU リソースを占有してしまい，他のノード更新が滞ってしまう．ノード更新処理が滞り，一連のノード更新処理全体にかかる時間が大幅に増加すると，システム全体の応答性 (responsiveness) が悪化してしまう．この現象のように，リアクティブな振る舞いのための一連の計算処理中に「重たい」計算処理が混じることで，システム全体の応答性が悪化してしまう問題はリアクティブプログラミングの文脈において **Reactive Thread Hijacking Problem (RTHP)** [72] と呼ばれている．

本論文では，**毎回の時変値更新処理のたびに処理の実行が要求されるわけではないが，最悪実行時間が比較的長くなってしまい RTHP を引き起こすことがある計算処理を Heavy-task** と呼ぶ．**Heavy-task** は計算に時間がかかる CPU バウンドな処理を対象とし，特に複雑なデータ構造への書き込みや読み出しを行う処理のことを指す．通信や外部イベント待ちのために時間がかかる IO バウンドな処理は対象としない．第 2.4.5 節で説明したように，Emfrp は言語機能的制限をもつため Heavy-task を Emfrp の式で直接的に実装することは難しい．第 3 章にて提案した $\text{Emfrp}^{\text{BCT}}$ では，リスト構造や木構造程度の再帰データ型の使用が可能であるため，例えばリスト操作を行うような Heavy-task を直接的に実装することが可能である．しかしながら，$\text{Emfrp}^{\text{BCT}}$ はデータ構造のインプレースな更新には対応していないため，Heavy-task の最中にデータのコピーが頻繁に発生しメモリ使用量が増大する傾向にある．そのため，$\text{Emfrp}^{\text{BCT}}$ でのプログラム記述では RTHP とメモリ使用量が増加する問題が同時に発生する．FFI(Foreign function interface) を用いて C 言語の関数やデータ構造として Heavy-task を実装することで，後者の問題を部分的に解決することができる．C 言語上のデータ構造ならばインプレースな更新が行えるため，メモリを不必要に使用することはない．しかし，ノード更新処理において，C 言語上のデータ構造と FRP 言語上のノードの状態の整合性をうまく保つ必要がある．また，単純に FFI を使用して Heavy-task を実装しても前者の問題である RTHP の解決には寄与しない．RTHP の本質的な解決には，リアクティブな振る舞いと Heavy-task による高負荷な計算とを協調的に動作可能な非同期実行機構が必要となる．

4.2.2 ケーススタディ：マイクロマウスロボット

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」 [*1] の迷路探索走行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．

[*1] `https://en.wikipedia.org/wiki/Micromouse`

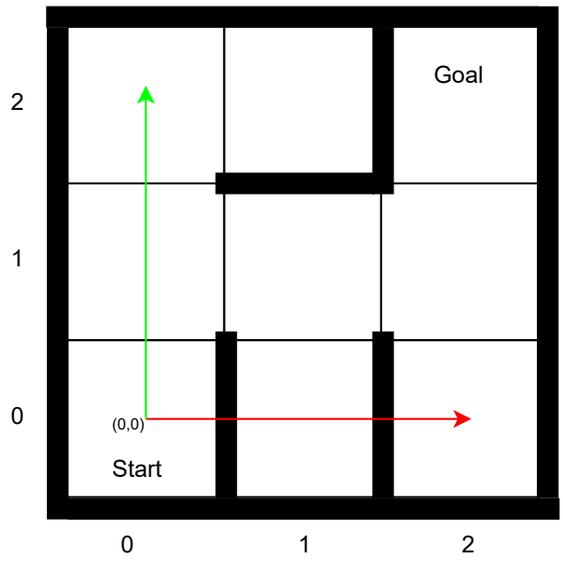

図 4.1　迷路の例

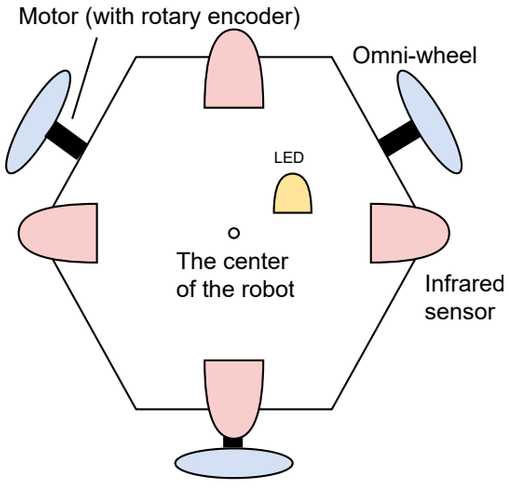

図 4.2　迷路探索ロボットの概要図

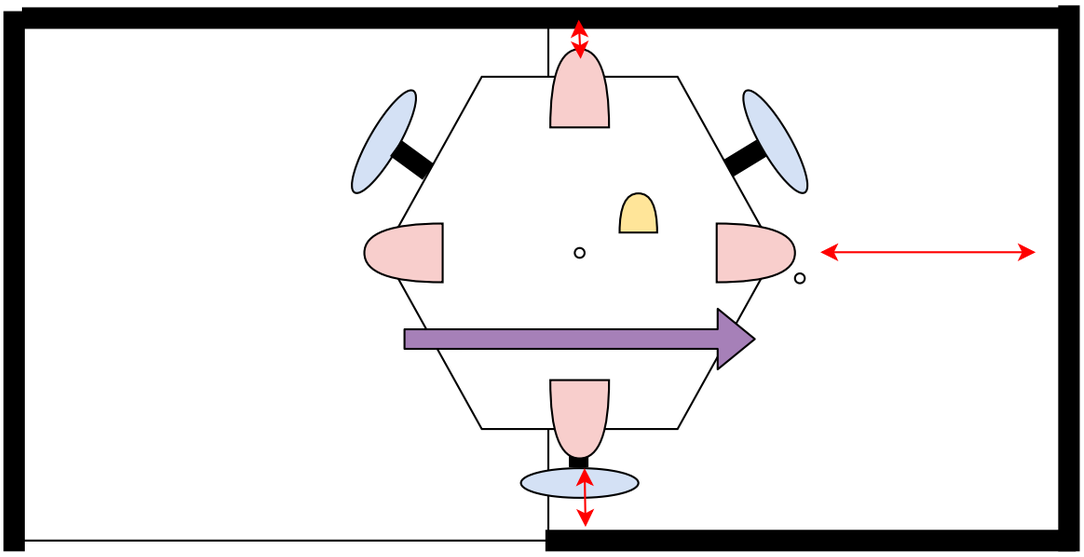

図 4.3　ロボットが左の区画から目標区画 (右の区画) へと進入する様子

問題設定

迷路の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 4.1 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．

ここでは簡単のため，ロボットは全方向移動車とする．[*2]ロボットを上部から見た概要図を図 4.2 に示す．ロボットには壁の有無および壁との距離を判断するための赤外線センサー (Infrared sensor) が 4 つ，オムニホイール (Omni-wheel)[*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 4.2 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，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．

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．

これら一連の動作は，以下の 3 つの動作に分解することができる．

1. ロボットを目標区画へ移動させる
2. 未知の壁情報を記録する
3. 記録した壁情報を迷路探索アルゴリズムへの入力とし，次に向かうべき区画を計算する

動作 1 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，「モーターへの出力比率」を出力としたリアクティブな振る舞いである．一方で，動作 2 と動作 3 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，それぞれを Heavy-task と見なすことができる．

前述したロボットの動作では目標区画への移動 (動作 1)，壁情報の記録 (動作 2)，次の目標区画の設定 (動作 3) がそれぞれ順番に行われていた．しかし，図 4.3 に示すように，ロボットの機体中心が目標区画の中心に十分に近い場合には，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可能である．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

[*2] マイクロマウス競技者の多くは独立 (差動) 二輪車による機体で競技に参加している．独立二輪車の場合には，例えば隣の区画に移動する際に車体の方向を管理し，直進動作と回転動作を適切に切り替える必要がありプログラムが煩雑になる．一方で，モーターが 2 つで十分なこと，特殊な車輪を必要としないことから車体の小型軽量化が可能であり，走行時間の短さを競う競技ルールとしては独立二輪車の方が有利である．

[*3] https://en.wikipedia.org/wiki/Omni_wheel

```
module MovePID # Position controller
in tar_x : Float, tar_y : Float, # Target
   enc1: Int, enc2: Int, enc3: Int,
   duration : Float # Elapsed time [sec]
out distance : Float,
    duty1 : Int, duty2 : Int, duty3 : Int
use Std

# Coefficients of PID
data (Kp, Ki, Kd) = ...
# From motor rpm to movement vector.
func motor_to_vector(...) = ...
# From movement vector to power ratio.
func motor_speed(...) = ...

# updating current positions.
node (mx, my) = motor_to_vector(enc1, enc2, enc3)
node init [(0, 0)] (cur_x, cur_y) = (cur_x@last + mx, cur_y@last + my)

# PID control for X axis.
node dx = Kp * e_x + Ki * ei_x + Kd * ed_x
node init [0] e_x = tar_x - cur_x
node init [0] ei_x = ei_x@last + (e_x + e_x@last) / 2 * duration
node ed_x = if duration == 0 then 0 else (e_x - e_x@last) / duration

# PID control for Y axis (same as X).
node dy = Kp * e_y + Ki * ei_y + Kd * ed_y
node (e_y, ei_y, ed_y) = ...

# Distance to target
node distance = sqrt(e_x*e_x + e_y*e_y)
# Output to motors
node (duty1, duty2, duty3) = motor_speed(dx, dy)
```

図 4.4　PID 制御による機体位置の制御器

画へと進行することができる．これによりロボットが迷路を探索している時間が短縮する．これは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ことを示唆している．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．

Emfrp による迷路探索ロボットの動作記述

前述したロボットの制御プログラムを Emfrp で記述していく．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 4.4 に示す．このモジュールは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．機体座標についての 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 4.5 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 4.6 に示す．

```
1  module MazeRunnerEmfrp
2  in sn : Int, se : Int, ss : Int, sw : Int,
3     enc1: Int, enc2: Int, enc3: Int,
4     time(0) : Float, # Time from start [sec].
5     to_u : Int, to_v : Int
6  out finished : Bool,
7      duty1 : Int, duty2 : Int, duty3 : Int,
8      reached_target : Bool,
9      wall_n : Bool, wall_e : Bool,
10     wall_s : Bool, wall_w : Bool
11 use Std
12
13 data G_U = ... # X-index of goal section.
14 data G_V = ... # Y-index of goal section.
15 data WS = ... # The width of sections.
16 data THR_WALL = ... # Threshold of sensors.
17 data THR_TARGET = ... # Epsilon distance.
18
19 # Time taken for the previous iteration.
20 node duration = time - time@last
21
22 # Control position.
23 newnode target_dist, d1, d2, d3 =
24   MovePID(to_u * WS, to_v * WS, enc1, enc2, enc3, duration)
25
26 node duty1 = if reached_target then 0 else d1
27 node duty2 = if reached_target then 0 else d2
28 node duty3 = if reached_target then 0 else d3
29
30 # Whether the robot reached target section.
31 node reached_target = target_dist < THR_TARGET
32
33 # Wall information
34 node wall_n = sn > THR_WALL
35 node wall_e = se > THR_WALL
36 node wall_s = ss > THR_WALL
37 node wall_w = sw > THR_WALL
38
39 # Whether the robot reached goal.
40 node finished = (to_u == G_U && to_v == G_V && reached_target)
```

図 4.5　Emfrp による迷路探索ロボットのプログラム

`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．インデックスは座標ではなく，迷路を 2 次元配列と見立てた時の添字を表す．図 4.1 の迷路では，スタート地点は $(0, 0)$，ゴール地点は $(2, 2)$ である．図中の赤矢印は水平方向 X の軸，緑矢印は垂直方向 Y の軸を表し，インデックスは (X, Y) で表記される．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．26–28 行目は，目標区画に到着した場合には，それ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 4.6 では，入力関数 (`Input`) と出力関数 (`Output`) にまたがるグローバル変数 (`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 4.6，24 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

```c
// Global variables across Input and Output
int next_u = 0; // X-index of the next target
int next_v = 1; // Y-index of the next target
bool has_next = false;

void Input(int* sn, int* se, int* ss, int* sw,
  int* enc1, int* enc2, int* enc3,
  float* time, int* to_u, int* to_v){
  get_sensors(sn, se, ss, sw);
  get_encoders(enc1, enc2, enc3);
  get_time(time);
  if(has_next){
    *to_u = next_u; *to_v = next_v;
    has_next = false;
  }
}

void Output(bool finished,
  int duty1, int duty2, int duty3,
  bool reached_target, bool wall_n,
  bool wall_e, bool wall_s, bool wall_w){
  if(finished) set_led();
  set_motor(duty1, duty2, duty3);
  if(reached_target){
    // Heavy task executions
    set_wall(next_u, next_v,
             wall_n, wall_e, wall_s, wall_w);
    calc_next(&next_u, &next_v);
    has_next = true;
  }
}
```

図 4.6　`MazeRunnerEmfrp`モジュールの入出力関数 (C 言語)

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するため，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る．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．つまり，Emfrp のような Heavy-task との協調機構を持たない push 型 FRP 外部 DSL による記述では以下の 2 点が問題となる．

1. Heavy-task の**直線的な実行**の影響が，リアクティブな振る舞いに波及し応答性が悪化する．
2. 出力から入力に対する依存関係や Heavy-task が対象としているデータ構造が FRP 言語内 (Emfrp プログラム中) で**暗黙的に出現**し明記されることはない．

問題 1 からナイーヴな実装では改善探索を実装することは困難である．問題 2 から Heavy-task と協調するプログラムでは，データの流れが FRP プログラムと C 言語コードに分散しすることでコード理解や変更を妨げる設計となる．

Emfrp と RTOS との協調動作

問題 1 に対処しつつ改善探索を実装するために，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することを考える．この場合，Emfrp のイテレーションと並行に Heavy-task(迷路情報の更新と探索アルゴリズムの適用) を実行するような実装となるだろう．シングルコアマイコン上での実行を想定すると，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多少長くなるが，長時間停止することはない．そのため，スケジューリングに関するパラメータを適切に設定すれば応答性に問題はないと考えられる．したがって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になる．

しかし，Heavy-task 実行は全て C 言語 (RTOS 上のタスク) で行われてしまうため，問題 2 は依然として解決できていない．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整合性を保ちつつ行う必要があり，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 等の FRP に特化した外部 DSL を使用する利点が損なわれてしまう (問題 3).

4.3 Emfrp$^{\mathrm{HT}}$

問題 1，問題 2，問題 3 のそれぞれを解決しつつ，リアクティブな振る舞いと Heavy-task 実行を両立するために，Emfrp に対して言語拡張およびランタイム拡張を加える．拡張された Emfrp を Emfrp$^{\mathrm{HT}}$(Emfrp with Heavy Task) と呼ぶ．本節では，Emfrp$^{\mathrm{HT}}$ によって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．

4.3.1 言語拡張

Emfrp$^{\mathrm{HT}}$ で記述した迷路探索ロボットの例を図 4.7 に示す．記述例中には導入された言語機構が含まれる．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 4.8 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.

新たに導入された言語機構を以下に挙げる．

- 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 4.8，14–25 行目)
- モジュール間でタスクリソースを受け渡す機構 (図 4.7，3 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 4.7，15–17 行目，20–21 行目)

タスクとタスクリソース

タスクとタスクリソースは図 4.8，14–25 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る．タスクリソースはタスクによって扱われるデータの抽象化であり，実行時にはタスクリソースインスタンスがタスク内部で操作される．タスクリソース定義とタスクリソースインスタンスの関係は Java(や他の OOP 言語) でのクラスとインスタンスの関係と同等のものである．

タスクは次の構文によって定義される．

$$\texttt{task}\ g\texttt{:(}x_1 : \tau\texttt{,...,}x_n : \tau\texttt{)->(}y_1 : \tau\texttt{,...,}y_m : \tau\texttt{)/}p$$

g はタスク名，x_i は入力の名前，y_i は出力の名前，τ 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

```
# MazeRunner.mfrp
module MazeRunner
in mg : resource MazeGraph,
   sn : Int, se : Int, ss : Int, sw : Int, enc1: Int, enc2: Int, enc3: Int, time(0) : Float
out finished : Bool, duty1 : Int, duty2 : Int, duty3 : Int
use Std, Resources

node duration = time - time@last
newnode target_dist, duty1, duty2, duty3 = MovePID(to_u * WS, to_v * WS, enc1, enc2, enc3, duration)

# Whether the robot has neared the target section
node init[False] near_target = target_dist < SECTION_WIDTH * 0.4
node reached_target = target_dist < THR_TARGET # Whether the robot reached target section.

tasknode finish_register : Future[Unit] =  # Register section information
  RegisterSection(to_u, to_v, wall_n, wall_e, wall_s, wall_w) with mg
    at (!near_target@last && near_target)  # positive edge of near_target

# Search result (if it has already reached the goal, keeps returning the goal coordinates)
tasknode next : Future[(Int, Int)] =
  CalcNextSection(to_u, to_v, G_U, G_V) with mg at wait(finish_register)

# Which direction the robot is going
node move_dir = (to_u - from_u, to_v - from_v) of:
  (0, 0) -> Stop,
  (0, 1) -> StoN, (0, -1) -> NtoS, (1, 0) -> WtoE, (-1, 0) -> EtoW,
  _ -> Stop # unreachable

node wall_n = move_dir of:  # Whether there is a wall to the north in target section
  Stop -> sn > THR_WALL_SHORT,
  StoN -> sn > THR_WALL_LONG,
  NtoS -> False, # Absolutely no walls.
  WtoE -> sn > THR_WALL_SHORT,
  EtoW -> sn > THR_WALL_SHORT
node wall_e = ... # omitted
node wall_s = ...
node wall_w = ...

# Update target section (always move forward to (0, 1) at first)
# Define nodes of:
#  previous target (from_u, from_v), current target (to_u, to_v), temporarily next target (tmp)
node init[(0, 0, 0, 1, None)] (from_u, from_v, to_u, to_v, tmp) = (reached_target, next, tmp@last) of:
    (True,  Ready((nu, nv)), _) -> (to_u@last, to_v@last, nu, nv, None),
    (False, Ready(n),        _) -> (from_u@last, from_v@last, to_u@last, to_v@last, Some(n)),
    (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)

node finished = move_dir of:  # Whether the robot reached goal section
  Stop -> to_u == G_U && to_v == G_V && reached_target,
  _ -> False
```

図 4.7　Emfrp$^{\mathrm{HT}}$ による迷路探索ロボットのプログラム

```
# Resources.mfrp
material Resources;
type MoveDir = Stop | StoN | NtoS | WtoE | EtoW
data THR_WALL_SHORT = ... # Threshold for wall
data THR_WALL_LONG = ...  # Threshold for wall
data (G_U, G_V, WS, THR_TARGET) = ...

# Function to wait for task completion.
func wait(s : Future[A]) = s of:
  Ready(_) -> True,
  Pending -> False, NotStarted -> False

# Define task resources and the tasks.
resource MazeGraph {
  # Task to record wall information of section (u,v) in a MazeGraph instance
  task RegisterSection :
    (u: Int,  v: Int,
     n: Bool, e: Bool,
     s: Bool, w: Bool) -> (h: Unit) / write

  # Task to compute the next section (next_u,next_v) to go from (u,v) to the goal
  task CalcNextSection :
    (u: Int, v: Int, goal_u: Int, goal_v: Int)
      -> (next_u: Int, next_v: Int) / read
}
```

図 4.8　MazeRunnerモジュールのためのマテリアルファイル.

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 4.7，3 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 (4.3.2 節にて解説する).

Future型とタスクノード tasknode

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，本研究では以下に定義する Future型を導入する．

```
type Future[A] = Ready(A) | Pending | NotStarted
```

Aはタスクの結果を表す型を表す型パラメータである．Future型を持つノードは，関連したタスクが完了した直後のイテレーション中のみ Ready(_)の値をもつ．それ以外のイテレーション中は，Pendingか NotStartedの値をもつ．これは言語ランタイムによって管理される．Future型ノードが Pendingを持つ時，その関連したタスクは実行およびその完了を待っている状態 (タスク発行済みの状態) である．NotStartedは，タスクが実行待ちになっていない状態 (タスクが発行されていない状態) を表す．

Future型をもつノードは特殊な構文 tasknodeによって以下のように定義され，タスクと紐づけられる．

$$\texttt{tasknode}\ z\texttt{:Future[(}\bar{\tau}\texttt{,...,}\bar{\tau}\texttt{)]} = g\texttt{(}e_1\texttt{,...,}e_n\texttt{)}\ \texttt{with}\ r\ \texttt{at}\ e_t$$

z はノード名，g はタスク名，r はタスクリソースインスタンス，$e_1, \cdots, e_n$ はタスクへの入力式，e_t は

タスクの発行条件式である．タスク g は，ノード z が Pendingでない (すなわち Readyか NotStarted) のとき，かつ e_t が Trueへと評価されるときに**発行**される．タスクが発行されると，そのイテレーションでの入力式のスナップショットが保存され，タスクはタスク実行中にスナップショット中の値を参照することができる．

tasknodeに対してのノードの依存解析 (ループ検査) は行われない．タスク発行条件式が Trueかどうかの検査とタスクの発行は，全ての通常ノードの更新処理を行なった後に行われる．つまり，発行されたタスクがイテレーションの途中で完了し，関連した tasknodeが Readyになることはない．そのため，ノード定義の依存関係に関する不整合はおこらない．また，Future型をもつノードに対する@lastは通常ノードと同様に直前値 (1 イテレーション前の値) が参照される．

tasknodeの制限

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高々 1 つしか存在できない．例えば上記の例では，タスクリソースインスタンス mgを利用する RegisterSectionタスクについての tasknode定義は 16 行目で行われている．これに追加して，同じ mgを利用する RegisterSectionタスクに紐づく tasknode定義は禁止される．この制限は，言語ランタイムが最大でいくつのタスク実行を扱えば良いのかを静的に判断するために必要である．これにより Emfrp^{HT} の実行中には動的なヒープメモリ確保が必要ないため，予期せぬメモリ不足でシステム全体がクラッシュしてしまうことを避けられる．

4.3.2 ランタイム拡張

Emfrp^{HT} では，Emfrp の実行対象と同様に小規模で計算リソースの限られた組込み環境に向けてランタイムの拡張を行う．そのため，特にシングルコアマイコンでも動作するようなランタイム拡張を目指す．Emfrp の実行モデルに対し非同期タスク実行基盤を取り入れる際に，以下の 3 つの項目を目標とした．

- タスク間で共有されるタスクリソースについての整合性を保つこと．すなわち，それらが競合しないように言語ランタイムが管理する
- シングルコアマイコン上でも動作できること
- ヒープメモリの動的な確保が必要ないこと

タスクとタスクリソースの実行時表現

図 4.8 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 4.9 に示すテンプレートコードに変換される．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定数によってユーザーが指定する．4.3.1 節にて説明した tasknodeの制限から，プログラム中で発生しうるタスクの総数やタスクの唯一性が静的に決定されるため，言語ランタイムによる動的なメモリ確保は必要ない．これによって (目標 3) が達成される．

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 4.10 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をローカル変数 mgとして定義していることである．トップレベルモジュールの入力として指定されたタ

```c
// Resource.c
struct MazeGraph { /* ... */ };

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

#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) {
  /* Search next direction */
}
```

図 4.9　タスクリソース定義とタスク定義のコンパイル結果 (C 言語上での表現)

```c
// MazeRunnerMain.c
void Input(...){ /* ... */ }
void Output(...){ /* ... */ }
int main(void){
  /* Initialization of sensors */
  /* Initialization of other devices */
  /* Initialization of resource instance (mg) */
  struct MazeGraph mg = { /* ... */ };
  ActivateMazeRunner(&mg);
  return 0;
}
```

図 4.10　MazeRunnerモジュールの入出力関数定義 (C 言語のスケルトンコード)

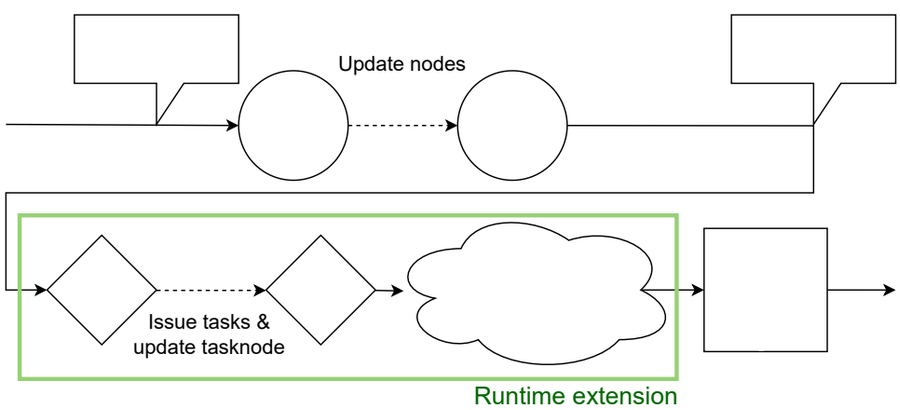

図 4.11　イテレーション処理の拡張

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．

Algorithm 2 タスクオブジェクト g^r (リソースインスタンス r を扱うタスク g) の発行と `tasknode` z の更新の擬似コード

```
1: function UPDATE_AND_ISSUE_g_WITH_r( )
2:     if g^r.pending then
3:         emit Pending to z
4:     else if eval(e_t) == True then
5:         v_1 ← eval(e_1); ...; v_n ← eval(e_n)
6:         g^r.parameter ← (v_1, ..., v_n)
7:         if W_r = ∅ ∧ MUTEX(g^r, E) then
8:             E.enqueue(g^r)
9:         else
10:            W_r.enqueue(g^r)
11:        end if
12:        emit Pending to z
13:        g^r.pending ← true
14:    else
15:        emit NotStarted to z
16:    end if
17: end function
```

イテレーションの拡張

2.4.3 節にて説明した Emfrp の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 (図 4.11)．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タスクの発行および非同期タスク実行が行われる．そのため，Emfrp$^{\mathrm{HT}}$ のイテレーションは 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必要がある．

非同期タスク実行機構

アルゴリズム 2 とアルゴリズム 3 が我々の提案する非同期タスク実行機構の核心である．我々の手法では，(目標 1) を守るため，タスク定義時に指定された `read`と `write`を処理系へのヒントとして利用する．実行基盤は，1 つの実行リスト E とリソースインスタンス r ごとの実行待ちキュー W_r もつ．それぞれのアルゴリズム中の g^r は，リソースインスタンス r を操作するタスク g を表現するオブジェクトである．実行の途中であるタスクは実行リスト E 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 W_r にエンキューされる．タスクオブジェクト g^r について，「(1)g が `read`タスクかつ E の中の r を対象にするタスクは全て `read`タスクである，または，(2)g が `write`タスクかつ E の中に r を対象にするタスクが存在しない」時に g^r を E に加えることができる．この条件を判定する述語を $MUTEX(g^r, E)$ とする．それぞれのアルゴリズムでは $MUTEX$ を使って，タスク実行の際のタスクリソースインスタンスについての排他制御を行なっている．

アルゴリズム 2 は，タスクの発行を行う関数である．この関数は図 4.11 のダイヤモンド形の部分でそれぞれの g と r ごとに実行される．アルゴリズム中の z, e_1,...,e_n,e_t は，g^r と対応した `tasknode` z `=` g`(`e_1`,...,`e_n`)` `with` r `at` e_t によるものである．eval(e) は時変値 e の評価を行う．g^r.pending はタスクオブジェクト g^r が実行キューあるいは実行待ちキューに含まれているかどうかを表す変数である．

Algorithm 3 非同期タスク実行の擬似コード

```
1: function EXECUTE_TASK( )
2:     if E = ∅ then
3:         return
4:     end if
5:     g^r ← E.dequeue()
6:     (is_done, result) ← ctxsw_N(g, r, g^r.parameter, g^r.stack)
7:     if is_done then
8:         emit Ready(result) to z
9:         g^r.pending ← false
10:        while W_r ≠ ∅ ∧ MUTEX(W_r.peek(), E) do
11:            y^r ← W_r.dequeue()
12:            E.enqueue(y^r)
13:        end while
14:    else
15:        E.enqueue(g^r)
16:    end if
17: end function
```

g^r.parameter はタスクの引数のスナップショットを保存する変数である．タスクの状態が pending でない場合に，発行条件を検査し，タスクの発行をおこなう．その際に $MUTEX$ を使用し E と W_r のどちらにタスクオブジェクトを挿入するか決定している．また，`tasknode`の値の更新も行う．

アルゴリズム 3 は，発行されたタスクを断続的に非同期実行する関数である．この関数は図 4.11 の雲形の部分で実行される．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 つのタスク実行が行われる．タスクの実行と休止には一般的な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．アルゴリズムの 6 行目はコンテキストスイッチによるタスク実行を表す疑似コードである．コンパイル時に与えられる時間 N だけタスク g にコンテキストスイッチして実行を行う．このときタスク関数には，リソースインスタンス r と保存されていたパラメータ g^r.parameter が渡される．また，その実行は静的に確保されたスタック領域 g^r.stack にて行われる．タスクを一定時間実行したのち，完了した場合には，対応した `tasknode`を `Ready`状態で上書きし，リソースインスタンス r の他のタスクを実行キュー E に挿入する．タスク実行がラウンドロビン方式であること，タスクは待ちキューの先頭から順番に実行キューに移動することから，このスケジューリングアルゴリズムでは飢餓状態 (発行されたタスクが実行されない状態が続く) は起こらない．このアルゴリズムは非常にシンプルかつ (特殊なハードウェアではない) タイマ割り込みを必要とするものであるため，(目標 2) を満たす．

4.3.1 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る．これにより，(目標 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] `https://www.absint.com/stackanalyzer/index.htm`

4.3.3 Emfrp^{HT} による例題の記述

図 4.7 に示した MazeRunnerモジュールの動作について説明する．まず，3 行目にてタスクリソース MazeGraphのインスタンス mgがモジュールに入力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，Emfrp での 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部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．9 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．

12 行目の near_target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，15–17 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 4.8 の 9–11 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．20–21 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 41 行目と 42 行目である．41 行目は機体が目標区画に到着した後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すぐに (to_u，to_v) に反映させる．42 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 43 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．4.2.2 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．

4.4 評価

Emfrp^{HT} では，リアクティブな振る舞い (ノード更新処理) と Heavy-task の非同期実行による協調によって，システムの応答性の悪化が抑制可能であることを，小規模な例を実装することで確認した．

4.4.1 問題設定

Emfrp^{HT} による応答性の悪化の低減を観察するために，評価実験を行なった．また，同時に Heavy-task 実行に関するオーバーヘッドを計測した．以下では，評価の対象としたプログラムおよびその実行環境，計測方法について説明する．

対象プログラム

評価の対象としたプログラムを，図 4.13 と図 4.15 に示す．図 4.13 は Emfrp による記述であり，その入出力コード (C 言語記述) は図 4.14 である．これと同等の目的で Emfrp^{HT} によって記述されたプ

```c
int tarai(int x, int y, int z) {
    if (x <= y) return y;
  return tarai(tarai(x - 1, y, z), tarai(y - 1, z, x), tarai(z - 1, x, y));
}
```

図 4.12　C 言語によるたらい回し関数 (竹内関数)

```
module SingleTask_Emfrp
in  input : Int,
    task_result(0) : Int

out task_issue : Bool,
    task_input_1 : Int,
    task_input_2 : Int,
    task_input_3 : Int,
    output_result : Int
use Std

node task_issue = input > 0
node task_input_1 = 12
node task_input_2 = task_input_1 / 2
node task_input_3 = 0

node output_result = task_result * 2
```

図 4.13　単一の Heavy-task 実行を繰り返すプログラム (Emfrp)

ログラムは図 4.15 である．その入出力コードは図 4.16 である．

これらのプログラムは，入力ノード `input`に 1 以上の値が入力された時に単一の Heavy-task を起動する．そして，Heavy-task の計算結果を加工 (2 倍) したのち，外部へと出力を行うことを繰り返す．今回の評価では，Heavy-task としてたらい回し関数 (あるいは，竹内関数) [*5]を利用した．図 4.12 に C 言語による実装を示す．

Emfrp によるプログラム (図 4.13) では，出力関数内で Heavy-task 実行を行うために，タスクを実行するかどうかの判定 (ノード `task_issue`) やタスクの入力 (ノード `task_input_1`，`task_input_2`，`task_input_3`) を出力ノードとして設定している．また，タスクの実行結果は，出力関数と入力関数にまたがった C 言語上のグローバル変数 (`tarai_result`) に格納されるそして，それが入力ノード (`task_result`) として設定されることによって Emfrp モジュール上で使用される．この`SingleTask_Emfrp`モジュール中には，Heavy-task の存在が明示されることはない．そして，Heavy-task の入出力間のデータフローがノードの入出力と逆転していることや，入出力関数にまたがるグローバル変数によって，見通しの悪いプログラムとなっている．また，出力関数内で Heavy-task 実行が行われるため，RTHP が発生する．

C 言語によって記述された入力関数 (図 4.14) では，評価実験のために 500 イテレーションごとに`input`に対して 1 を出力している．これによって，Heavy-task の実行は 500 イテレーションごとになる．出力関数内では，モジュールからの出力ノード`task_issue`の結果に応じて，Heavy-task であるたらい回し関数を実行する．この際に，Emfrp のイテレーションを中断しつつ，Heavy-task 実行が行われるため，RTHP が発生する．また，入力関数中や Heavy-task 実行中はマイコンの GPIO ピンを HIGH に設定する (GPIO ピンの使用用途については後述)．

Emfrp^{HT} によるプログラム (図 4.15) では，タスク定義およびタスクリソース定義によって，たら

[*5] `https://en.wikipedia.org/wiki/Tak_(function)`

```c
int tarai_result; // 結果共有のためのグローバル変数

void Input(int* input, int* task_result) {
  static int count = 0;
  // 実行時間計測のために GPIO に出力する
  HAL_GPIO_WritePin (GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
  count++;
  printf("INPUT\r\n");
  // 500イテレーションごとにタスク実行を要求する
  *input = (count % 500 == 0);
  *task_result = tarai_result;
  //  GPIO への出力を停止する
  HAL_GPIO_WritePin (GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
}

void Output(int* task_issue, int* task_input_1, int* task_input_2, int* task_input_3,
    int* output_result) {
  if(*task_issue){ // タスク実行すべきかどうか判定
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
    // Heavy-task 実行
    tarai_result = tarai(*task_input_1, *task_input_2, *task_input_3);
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);
  }
  printf("OUTPUT\r\n");
}
```

図 4.14　SingleTask_Emfrpモジュールのための入出力関数

い回し関数 (タスク Tarai) が Heavy-task となっていることが明示されている (1–4 行目)．タスク実行は，tasknode記述によって設定されている (18–20 行目)．これらの記述により，タスクとモジュールの入出力関係の逆転が起こることなく，またタスクの存在が DSL のソースコード中に明記されることで，データフローの見通しが改善されている．さらに，Emfrp^{HT} ではイテレーション毎にノード更新処理につづけて Heavy-task の分割実行が行われる．これによって，リアクティブな振る舞いと Heavy-task の並行実行が実現され，RTHP の発生を低減させることができる．今回の評価では，一度の Heavy-task 実行は連続して 10 ミリ秒おこなうとした．

Emfrp^{HT} のためのスケルトンコードは入出力関数の他に，タスクリソースの定義 (C 言語では構造体として定義される) と Heavy-task 実行用の関数定義が要求される．図 4.15 はこれらを含む C 言語コードである．入力関数にて，1000 イテレーションごとにタスク実行を促している．今回の計測に際して，Heavy-task 用関数には 1024 ワード (4096 byte) 分のスタック領域を割り当てた．また，入力関数内や Heavy-task 実行中には GPIO 出力を設定している．

実行環境

プログラムの実行には，ST マイクロ社の MCU，STM32F401RE を搭載したマイコンボード NUCLEO-F401RE [*6]を使用した．STM32F401RE は，表 2.1 に提示した性能である．今回の評価実験では，動作クロックを 84MHz に設定した．Emfrp^{HT} によるプログラムは，Heavy-task の分割実行のためにタイマ割り込みを使用する．タイマ 2 を 10kHz のカウントアップタイマとして設定して使用した．また，前述したとおり入力関数の実行中と Heavy-task の実行中には GPIO ポート B の 4 番ピンと 5 番ピンがそれぞれハイレベルになるように設定される．

プログラムのコンパイルおよびリンクには ST マイクロ社が提供する統合開発環境 STM32CubeIDE(Version 1.14.0) に付属するコンパイラ (arm-none-eabi-gcc) を使用した．最

[*6] https://www.st.com/en/evaluation-tools/nucleo-f401re.html

```
resource HT { # タスクリソース定義
  # タスク定義（たらい回し関数）
  task Tarai(x: Int, y: Int, z: Int) -> (result : Int) / read
}

module SingleTask_EmHT
in  tt : resource HT, # タスクリソースインスタンス入力
    input : Int

out output_result : Int
use Std

node task_issue = input > 0
node task_input_1 = 12
node task_input_2 = task_input_1 / 2
node task_input_3 = 0

# タスク実行の設定とノードへの紐づけ
tasknode result_f : Future[Int] = Tarai(task_input_1, task_input_2, task_input_3)
                                  with tt at task_issue

node init[0] task_result = result_f of
  Ready(r) -> r,
  _ -> task_result@last

node output_result = task_result * 2
```

図 4.15　単一の Heavy-task 実行を繰り返すプログラム (Emfrp$^{\mathrm{HT}}$)

```
// タスクリソースの定義（今回は空）
struct TaskResource_HT{};

// Heavy-task 実行関数
#define STACK_SIZE_Tarai 1024
void Tarai(struct TaskResource_HT* tt,
  int x, int y, int z,
  int* result){
    // 実行時間計測のために GPIO に出力する
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
    *result = tarai(x, y, z);
    HAL_GPIO_WritePin (GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);
}

void Input(int* input) {
  static int count = 0;
  // 実行時間計測のために GPIO に出力する
  HAL_GPIO_WritePin (GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
  count++;
  printf("INPUT\r\n");
  *input = (count % 1000 == 0);
  HAL_GPIO_WritePin (GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
}

void Output(int* output_result) {
  printf("OUTPUT\r\n");
}
```

図 4.16　SingleTask_EmHTモジュールのための入出力関数と Heavy-task 用関数

適化オプションは`-Os`によるサイズ最適化を指定した．実験対象の Emfrp プログラム (図 4.13) は，既存の Emfrp コンパイラによって C 言語コードへと変換した．これと C 言語による入出力関数記述および STM32F401RE のための周辺コードをリンクし，実験のためのバイナリファイルを得た．Emfrp^{HT} によるプログラム (図 4.15) は，著者が C 言語コードに手動で変換した．これと C 言語による入出力関数記述，コンテキストスイッチによるタスク実行ライブラリ，STM32F401RE のための周辺コードをリンクし，バイナリファイルを得た．タスク実行ライブラリは，著者らが作成したものである．このライブラリは，Arm Cortex-M4F(STM32F401RE に搭載されている CPU) の呼び出し規約および例外ハンドリングに則り，タイマー割り込みを使用して，プリエンプティブな時分割関数実行を提供する．

計測方法

評価対象のプログラムにおいて，GPIO ピンのレベル (振る舞い) を計測することで，入力関数と Heavy-task の実行時間の関係を測った．SIGLENT 社のオシロスコープ SDS1104X-E によって，GPIO 出力の波形を計測し，その概形を画像として出力した．

また，GPIO 出力の立ち上がりエッジの時間間隔などを計測するために，FPGA によるサンプリングを行った．FPGA による計測は，結果の解析，集計を容易にする目的がある．Digilent 社の FPGA，Zybo Z7 によって GPIO 出力を計測した．サンプリング周期は 50MHz のサイクルを 16 分周した 3.125MHz で行った．計測に使用した FPGA ソフトウェアおよび集計ソフトウェアは Github 上に公開している [*7]．

4.4.2 計測結果と考察

評価対象のプログラムそれぞれについて，応答性，時変値更新間隔，オーバーヘッドについての計測を行い両者を比較した．以下ではそれぞれの項目の結果を示しながら，考察をおこなう．

応答性

評価対象のプログラム 2 種では，入力関数の実行中にはポート B の 4 番ピンを，Heavy-task の実行中にはポート B の 5 番ピンをハイレベルに設定し，それ以外の部分ではローレベルに設定した．Emfrp のイテレーション実行の設計により，入力を得た後，すぐに時変値 (ノード) 更新処理，結果の出力，メモリ管理処理が続けて行われ，再度システム入力が行われる．すなわち，ポート B の 4 番ピンの立ち上がりエッジの間隔を計測することで，どの程度の頻度でシステム入力を受け付けているかを判断できる．この頻度が十分に高ければ，システムの応答性が担保されていると見なすことができる．また，同時にポート B の 5 番ピンを観察することで，Heavy-task 実行とイテレーション間隔の関係を明らかにすることができる．本項目の実験はオシロスコープによって，ポート B の 4 番ピンと 5 番ピンを同時に観察することでおこなわれた．

図 4.17(横軸:1s/div) に Emfrp プログラム (図 4.13) についての計測結果を提示する．以降のオシロスコープのキャプチャ画像での縦軸は 500mV/div である．図中の CH1(黄色の信号，ポート B の 5 番ピン) は Heavy-task の実行時間を表す．CH2(ピンクの信号，ポート B の 4 番ピン) は入力関数の実行時間を表す．さらに，Heavy-task 実行の切り替わり部分での拡大図をそれぞれ図 4.18(横軸:1ms/div) と図 4.19(横軸:1ms/div) に示す．CH1 の立ち上がりは Heavy-task の実行開始を表し，立ち下がりは実行終了を表す．Heavy-Task が実行されていない部分では，CH2 は矩形波の様相をしている．すなわち，イテレーションループが連続的に行われていることが確認できた．また，Heavy-task 実行中には通常イテレーションは全く行われずシステム入出力や内部状態の更新が全く行われないこと，すなわち RTHP が発生していることも確認できた．

[*7] `https://github.com/psg-titech/yokoyama-measurement-program`

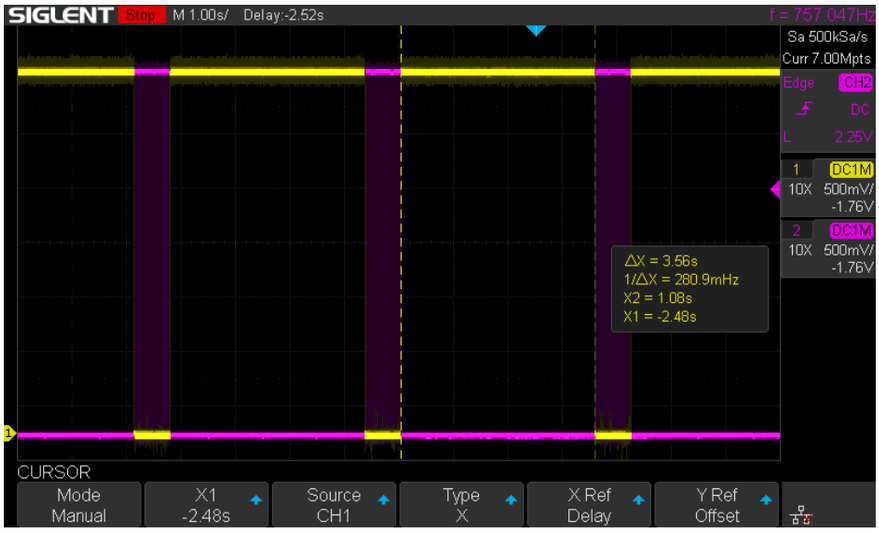

図 4.17　`SingleTask_Emfrp`モジュール実行時の時変値更新と Heavy-task 実行の様子 (全体図)

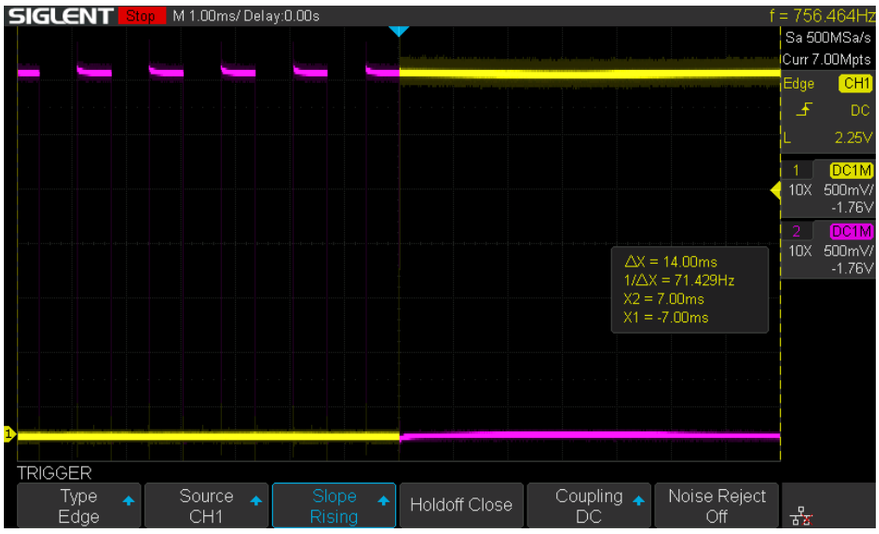

図 4.18　`SingleTask_Emfrp`モジュールにおける Heavy-task 実行開始の様子

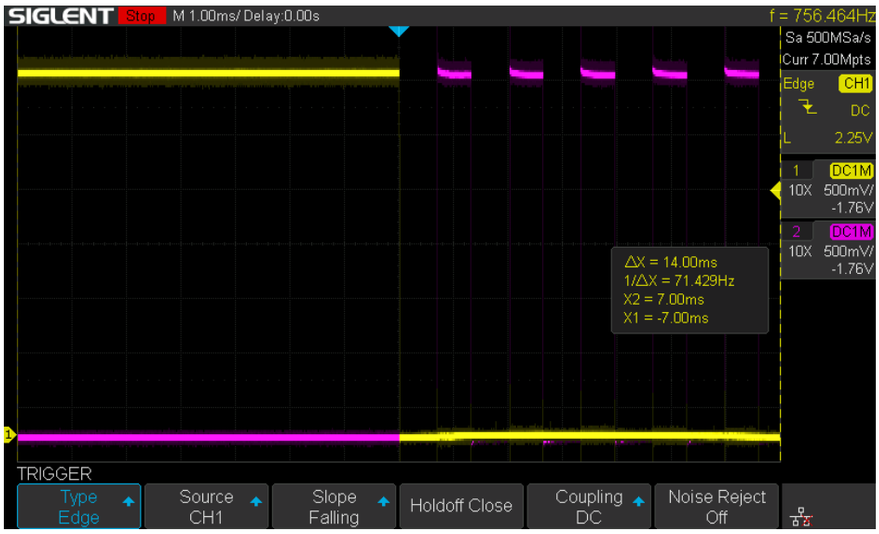

図 4.19　`SingleTask_Emfrp`モジュールにおける Heavy-task 実行終了の様子

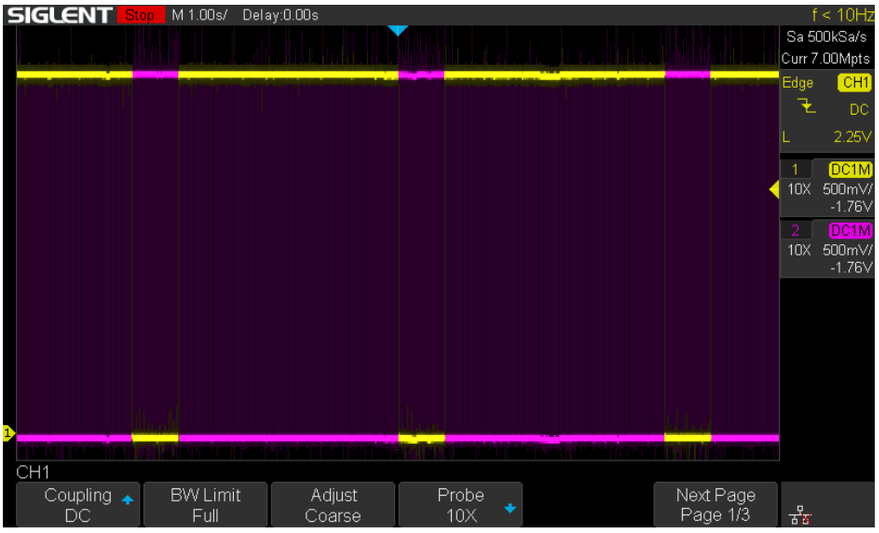

図 4.20　`SingleTask_EmHT`モジュール実行時の時変値更新と Heavy-task 実行の様子 (全体図)

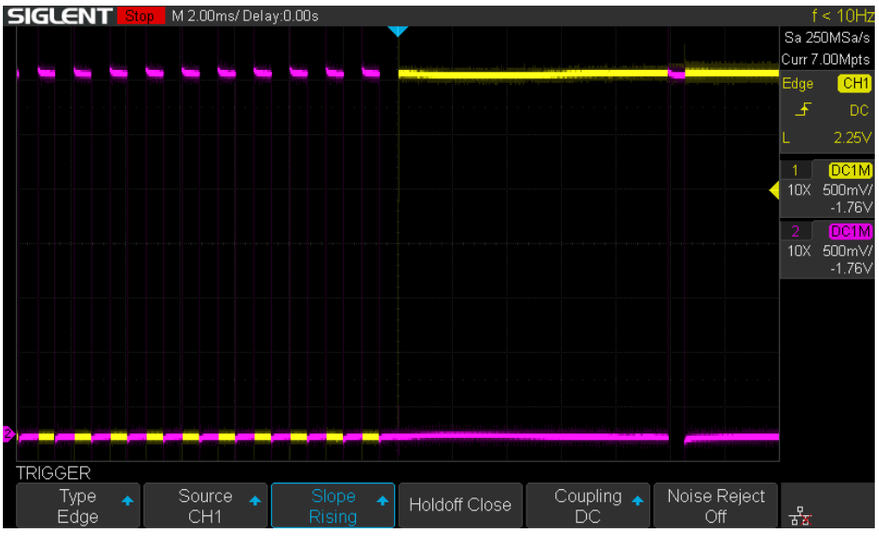

図 4.21　`SingleTask_EmHT`モジュールにおける Heavy-task 実行開始の様子

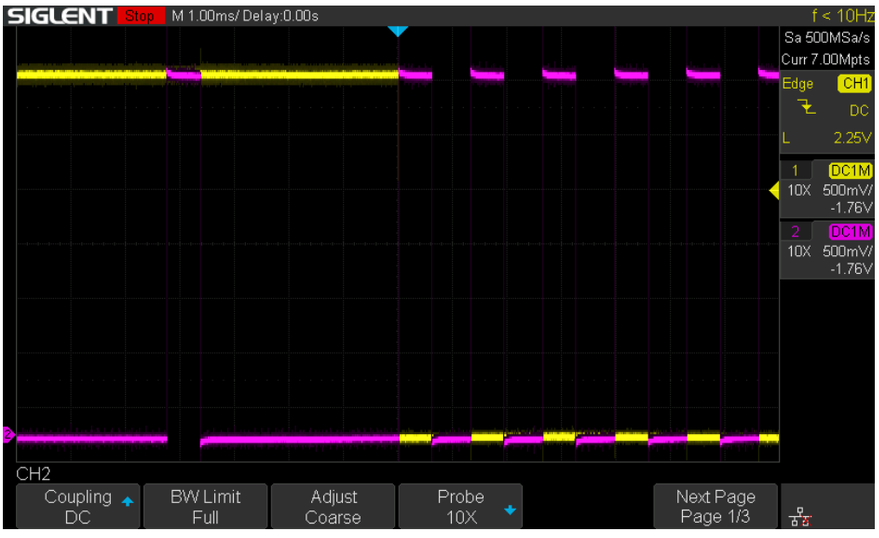

図 4.22　`SingleTask_EmHT`モジュールにおける Heavy-task 実行終了の様子

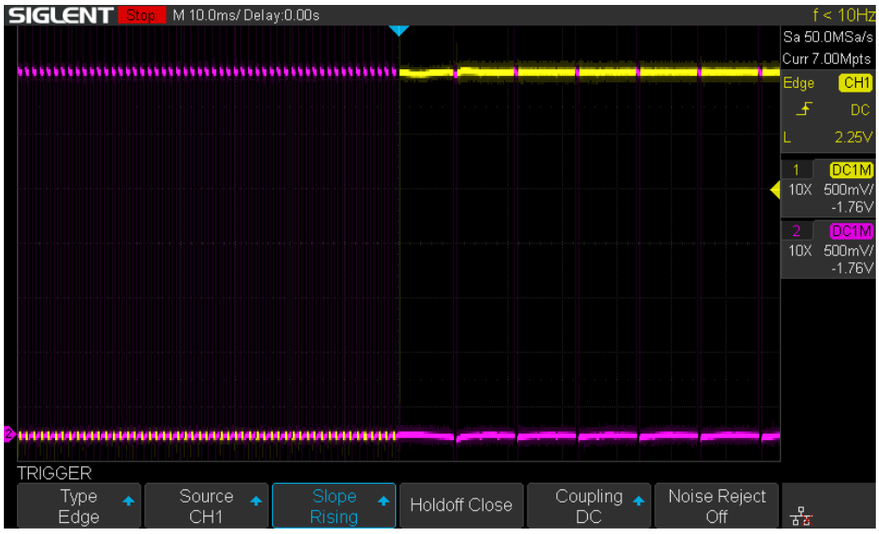

図 4.23　図 4.21 を俯瞰した図

次に図 4.20(横軸:1s/div) に Emfrp^{HT} プログラム (図 4.15) についての計測結果を提示する．信号 CH と GPIO ピンの対応は Emfrp のものと同様である．この図は図 4.17 と類似した概形をしているものの，測定範囲の全域にわたって薄くピンクの網掛けが表示されていることに注意されたい．すなわち，Heavy-task 実行中 (CH1 がハイレベルの間) にも，CH2 が度々ハイレベルになっている．Heavy-task 実行の切り替わり部分での拡大図をそれぞれ図 4.21(横軸:2ms/div) と図 4.22(横軸:1ms/div) に示す．また，図 4.21 を俯瞰した図を図 4.23(横軸:10ms/div) に示す．これらの測定結果から，Emfrp^{HT} で記述されたプログラムでは，Heavy-task 実行の途中にも定期的に時変値更新処理が行われていることがわかる．さらに図 4.23 では，Heavy-task 実行の前後でイテレーション間隔にどれだけの差異があるのか，より詳細に図示することができた．特に，Heavy-task 実行中のイテレーション間隔が約 10 ミリ秒であることが確認でき，10 ミリ秒間に設定した Heavy-task の連続実行時間との整合性が確認できた．

以上の測定結果から，Emfrp^{HT} で記述されたプログラムにおいて，RTHP による応答性の低下が軽減されていることが確認できた．また，Heavy-task の連続実行時間の設定と Heavy-task 実行中のイテレーション間隔の対応が確認できた．

時変値更新間隔

Emfrp や Emfrp^{HT} では，時変値更新処理は入力関数を起点にして行われる．したがって，ポート B の 4 番ピンの立ち上がり間隔を計測することで，イテレーションにかかる時間を測定することができる．評価対象プログラムのそれぞれについて，ポート B の 4 番ピンを FPGA を使用してサンプリングした．

表 4.1 に計測結果を提示する．表中の「Heavy-task 実行なし」は 2 つの Heavy-task 実行の間での通常イテレーションの時間を表す．「Heavy-task 実行あり」は Heavy-task 実行中に並行に行われるイテレーションの時間を表す．また，「Heavy-task 実行あり」のサンプルは Heavy-task 実行中に行われたイテレーションを全てサンプリングしたものである．そのため，サンプル数と Heavy-task 実行中に発生したイテレーション回数は一致している．Emfrp によるプログラムでは，Heavy-task 実行中にイテレーションは発生しない (正確にはイテレーション中に Heavy-task が実行される) ため Heavy-task 実行ありのサンプル数を 0 としている．Heavy-task 実行なしでの計測時間は，Emfrp によるプログラムでも Emfrp^{HT} によるプログラムでもほとんど差は出なかった．すなわち，Emfrp^{HT} によって管理されている Heavy-task が 1 つの場合には，イテレーション中の時間オーバーヘッドはほとんど計測されなかったと解釈できる．

表 4.1　イテレーションにかかる時間

言語 *1	最小 [us]	最大 [us]	標本平均 [us]	不偏分散	信頼区間 *2[us]	サンプル数
Heavy-task 実行なし (タスク実行完了から開始まで)						
E	1312.320	1322.880	1321.584	0.816	[1321.505, 1321.663]	499
H	1319.360	1321.920	1320.643	0.238	[1320.605, 1320.680]	639
Heavy-task 実行あり (タスク実行開始から完了まで)						
E	-	-	-	-	-	0
H	4255.680	11231.360	11202.096	134423.046	[11164.148, 11240.045]	361

*1 プログラム記述に使用した言語 (E:Emfrp(`SingleTask_Emfrp`)，H:Emfrp$^{\mathrm{HT}}$(`SingleTask_EmHT`))
*2 t 分布における平均値の 95% 信頼区間

表 4.2　Heavy-task 実行の時間オーバーヘッド

言語 *1	最小 [ms]	最大 [ms]	標本平均 [ms]	不偏分散	信頼区間 *2[ms]	サンプル数
E	3564.959	3569.641	3567.360	1.189	[3567.186, 3567.534]	153
H	4039.940	4046.252	4043.401	1.668	[4043.184, 4043.619]	138

*1 プログラム記述に使用した言語 (E:Emfrp(`SingleTask_Emfrp`)，H:Emfrp$^{\mathrm{HT}}$(`SingleTask_EmHT`))
*2 t 分布における平均値の 95% 信頼区間

表 4.3　セクションサイズの比較

モジュール名 (記述言語)	text [byte]	data [byte]	bss [byte]	合計 [byte]
`SingleTask_Emfrp` (Emfrp)	7800	166	2044	9960
`SingleTask_EmHT` (Emfrp$^{\mathrm{HT}}$)	9976	152	6344	16472

つぎに「Heavy-task 実行あり」では，イテレーションの平均時間が約 11 ミリ秒であることを確認できた．「Heavy-task 実行なし」でのイテレーションが約 1 ミリ秒かつ，Heavy-task の連続実行時間を 10 ミリ秒と設定したことと整合している．使用したタスク実行ライブラリでは，設定された時間を超えずにタスク実行を終了させるために，ユーザーから入力されたカウントから 1 を引いた値をタイマーの上限値として設定している．そのため，連続したタスク実行は 10 ミリ秒未満の時間で打ち切られる．このため，予測よりもイテレーションの平均実行時間が短くなっている．また，イテレーション時間の最小値が約 4 ミリ秒であるのは，タスク実行が早期に終了したためであると考えられる．

Heavy-task 実行の時間的オーバーヘッドを示す．評価対象プログラムのそれぞれについて，ポート B の 5 番ピンの状態を FPGA でサンプリングし，ハイレベルである時間 (Heavy-task の実行開始から完了までにかかる時間) を計測した．計測結果を表 4.2 に示す．今回の計測で使用した Heavy-task はたらい回し関数であったため，それぞれの場合で実行時間にはほとんどばらつきが出なかった．前述の表 4.1 から Heavy-task 実行中のイテレーション回数を 361 回，通常のイテレーションにかかる時間を標本平均から 1321 マイクロ秒とすると，Heavy-task の実行中に行われるイテレーションの総時間は $1321 * 361 = 476881$ マイクロ秒 (約 477 ミリ秒) である．ここに，表 4.2 から Heavy-task の連続実行にかかる時間 (標本平均)3567 ミリ秒を加算すれば，4044 ミリ秒であるから，Emfrp$^{\mathrm{HT}}$(`SingleTask_EmHT`) における測定結果での標本平均値約 4043 ミリ秒とほぼ一致することが確認できる．したがって，Heavy-task の分割実行にかかる時間オーバーヘッドは，イテレーション処理によるものであった．

次に静的な空間的オーバーヘッドを示す．評価対象プログラムのそれぞれのバイナリファイルのセク

ションサイズを表 4.3 に示す．Emfrp^{HT} によるプログラムは，Emfrp に比べ text 領域と bss 領域が増加している．text 領域に関しては，アルゴリズム 2 とアルゴリズム 3 の実装に加え，タスク実行ライブラリの実装が追加されたため増加したものと考えられる．bss 領域に関しては，Heavy-task 用関数のスタック領域として，4096byte の配列を追加で用意していることが，大幅な増加の理由だと考えられる．ただし，Emfrp での出力関数内 Heavy-task 実行においても再帰関数 (たらい回し関数) 実行のためのメモリ領域が実行時に必要とされる．そのため，Heavy-task 用のメモリ領域によって bss セクションサイズが増加することは，動的 (実行時) に必要なメモリ領域を静的なメモリ領域にオフロードしていると解釈することができる．

スケジューリングについての性能評価

今回の評価実験では，Emfrp^{HT} のタスクスケジューリングの性能については計測を行わなかった．これは，本研究が RTHP の抑制およびソースコード中のデータフローの明示を目的としているため，スケジューラの性能については提案の範囲外であるためである．

4.5 関連研究

並行計算における **future**(あるいは **promise**) パターンは，非同期計算の結果取得を実際に必要とされるまで遅延させるデザインパターンである．Java，JavaScript，Rust,Scala など非同期計算をサポートする多くの言語で，このパターンは実装されている [10, 15, 16, 48]．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，Heavy-task 実行はまさに非同期処理であり，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 [4] 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，本研究での Heavy-task 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，手続き型言語でのリアクティブプログラミングライブラリ (例えば REScala [55], ReactJS [35], RxJS [70]) を対象としてこの問題を分析した．その結果，**Reactive Thread Hijacking Problem (RTHP)** を提唱するに至った．彼らはこの問題に対する解決策として，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 [72]．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 モデルでは，リアクターの出力ノードからアクターを経由して別のリアクターの入力ノードへフィードバックことによってアクターとリアクター間の協調が行われる．彼らは Actor-Reactor モデルのために設計された言語 Stella によって，単一の言語内でアクターとリアクターを定義している．そのため，アクターとリアクターの間の依存関係は Stella 言語で記述されたコードを読むことで理解でき，システム記述の可読性，保守性に影響はない．一方で，Emfrp に Actor-Reactor モデルを取り入れることを考えると，アクターは C 言語で記述された入力関数や出力関数に相当すると見做せる．しかし，これは 4.2.2 節の (問題 2) として議論した，二つの言語間に跨った出力から入力への暗黙的な依存関係の導入となりプログラムの見通しが悪くなってしまう．本研究では，FRP 言語内に非同期実行プリミティブ (`tasknode`定義や `Future`型) を陽に導入することで暗黙の依存関係の問題への解決を図った．

Lustre [20, 45] は，Emfrp における`@last`と類似したオペレータ `fby`を使ってプログラムを記述できる同期的データフロー言語である．Cohen らは，**future** を Lustre に導入することで，並行性，パイプライン化，ジッター制御を可能にした [9]．彼らの future プリミティブは Lustre コード同士の並行処

理をサポートしている．彼らは future プリミティブを含むコードから，future を取り除いても計算の結果 (意味) が保たれることを形式的に示している．一方で，本研究では，(時間のかかる) 外部関数呼び出しをシステム全体の応答性を悪化させることなく実現するために Emfrp^{HT} を提案した．Emfrp^{HT} では，Emfrp コード同士の並行処理機構は提供されていない．`tasknode`によって記述される計算処理は通常の Emfrp では記述しづらいものであるため，意味の保存には Cohen らとは別の議論が必要だと考える．

Hailstorm [58] は Arrowized FRP [41, 47] の記法に影響を受けた IoT アプリケーション向け FRP 言語である．Arrowized FRP での`&&&`オペレータや`>>>`オペレータによってシグナル関数 (時変値上の関数) を合成する．**Juniper** [25] は，Web アプリケーション向け FRP 言語 Elm [12] と同様の `foldp`オペレータによって時変値の時間方向への畳み込み処理を行う．これらの FRP 言語も前述した Lustre も構文的な表現は異なるものの，シングルスレッド上で時変値が順番に処理されるという Emfrp と同様の実行モデルをもつ．それゆえ，RTHP によって応答性の悪化が引き起こされる．本研究の提案手法は，これらの言語での RTHP への対処の一助となることが期待される．

4.6 むすび

Emfrp に対して, 非同期タスク実行機構を導入した新たな FRP 言語 Emfrp^{HT} を提案した．Emfrp^{HT} では，リアクティブな振る舞いと比較的計算に時間のかかる処理 Heavy-task の協調がシステム全体の応答性を急激に損なうことなく可能である．言語拡張として，非同期タスクの発行とその結果の取得のために `tasknode`定義や `Future`型を導入した．ランタイム拡張として，オペレーティングシステムにおけるコンテキストスイッチによるタスク切り替え機構に加え，シンプルなタスクスケジューラを導入した．タスクスケジューラはタスクリソースの使用状況に従ったタスク実行の排他制御を自動的に行う．タスク実行は通常の時変値更新のあとに断続的に行われる．

Emfrp^{HT} では，タスクが扱うリソースを大きく抽象的にタスクリソースとして定義している．タスクリソースの細分化によって，より粒度の細かい排他制御が可能になり，タスク実行の効率化を計れると予想する．また，現状では Heavy-task は計算バウンドな処理を想定している．IO バウンドな処理についての Heavy-task 対応は今後の課題である．

第 5 章

電力リソース：EvEmfrp/S

5.1 まえがき

本章の主な貢献は非同期タイミングを起点として，**時変値更新間隔を切り替える機構を持った FRP 言語 EvEmfrp/S を設計したこと**である．これにより，以下についても貢献した．

- EvEmfrp(後述) の時変値更新のタイミング注釈の表現力の制限のため，十分な消費電力削減が行われないことを具体例をともなって指摘した．
- 提案言語 EvEmfrp/S による記述では，EvEmfrp で記述した同等のプログラムよりもシステムスリープの長期化を実現できることから消費電力の削減の機会が増加することを示した．
- EvEmfrp/S が持つ動的な切り替え可能な更新タイミングや遅延更新タイミングを実現するための，コンパイル方式および言語ランタイムを提案した．
- ボタン押下判定を行うプログラム例について実機による評価を行い，提案手法による省電力効果を確認した．

2.4 節で提示したように，Emfrp は連続的なイテレーションにより，応答性を確保している．そのため，必要以上の頻度で時変値更新処理が行われてしまい，システムの消費電力が増大する傾向にあった．EvEmfrp [66] は時変値更新の頻度を注釈できる言語機構を Emfrp に対して導入することで，この問題の解決を試みた．EvEmfrp はユーザーによって注釈された更新間隔をもとにシステム全体の間欠実行を行い，時変値更新の頻度を下げている．注釈は，具体的には定数値による時変値更新周期の設定と割り込みタイミングによる時変値更新をサポートしている．しかしながら，注釈の表現力の不足により，例えば「ある非同期タイミングを起点として，一定時間後に処理を行う」振る舞いを直接的にコードで表すことができない．そのため，このような振る舞いに反応する EvEmfrp プログラムでは不必要な時変値更新が行われ，省電力効果を十分に発揮できない場合がある．

ここでは，新たな時変値更新のタイミング注釈と，そこで注釈された更新タイミングを適切に扱うための言語ランタイム設計を提案する．提案する注釈では，更新タイミングの動的な切り替えをサポートしているため，前述した EvEmfrp が苦手とする振る舞いを直接的にコード中に記述できるようになる．これにより，従来の EvEmfrp によるプログラムよりも消費電力を抑えたプログラム記述が可能になる．いくつかの例を通して提案手法の有用性を述べたのち，処理系実装に際しての方針と現状の制限事項について述べる．

本章で新たに提案する EvEmfrp/S は，バッテリーや環境発電によって駆動するデバイスの運用時間の長期化に貢献する．EvEmfrp/S によるシステム記述はエッジデバイスの長期間にわたる安定的なセンシングを実現する一助となるだろう．

本章の内容は文献 [79] にて発表済みである．加えて本章では，例題に対する実機上での評価実験を行い，その結果を提示した．

```
module BlinkLED    # module name
in  btn (False) : Bool '(0, 20)
    # state of button (init. val=False)
    # '(0, 20) : update in each 20ms from 0ms
out led : Bool '(0, 20)
    # LED output (the timing that of button)

# checking the debounced button state
node btn_on = btn@last[1] && btn

# turn on/off when the button is pressed
node init[False] led =
  if not(btn_on@last[1]) && btn_on then not(led@last[1]) else led@last[1]
```

図 5.1　ソフトウェア的チャタリング除去後にボタン押下判定を行う EvEmfrp モジュール

5.2　背景：間欠実行による消費電力削減

本節では，時変値の更新周期を明示的に指定することができる機能を持った，Emfrp の後続言語 EvEmfrp の概要とその実行モデルについて説明する．

5.2.1　Emfrp 実行モデルの改良

EvEmfrp は，Emfrp の実行モデルを消費電力の観点で改善するために開発された言語である．前述したように，Emfrp はイテレーションをある種のポーリング処理によって実現する．入力に反応し時変値を更新したのち，すぐに次のイテレーションが行われる．この実行戦略では，言語ランタイムとして複雑なプログラムコードが必要ないためプログラムメモリを節約できる利点がある．一方で，想定されるシステムへの入力頻度よりもはるかに多い頻度でイテレーションが走り，不必要な時変値更新が呼び出される．結果としてシステムの消費電力が増大してしまう傾向にある．

EvEmfrp では，プログラマーがシステムに対して時変値更新の頻度をアノテーションできる機構を導入することで，上記の問題の解決を試みている．EvEmfrp コンパイラは，アノテーションされた時間間隔 (**周期イベント**) をヒントに適切な時変値更新間隔を算出する．言語ランタイムは算出された周期イベント毎に更新処理を行い，それ以外の時間はシステムをスリープ状態へと移行させる．さらに，EvEmfrp は**非同期イベント** (割り込み) による one-shot な更新処理もサポートしている．これにより，必要なタイミングでのみ時変値更新を行うことで，消費電力の削減を図っている．EvEmfrp についても Emfrp と同様の言語機能的制限が課されることで，空間漏れや時間漏れを起こさずに時変値更新を行うことができる．

EvEmfrp プログラムは Emfrp プログラムと同様にコンパイラによって解析されたのち，C 言語プログラムへと変換され同時にユーザーのためのスケルトンコードが生成される．スケルトンコードには外部環境 (センサーやアクチュエータ) との入出力のためのコード，非同期イベントの割り込み設定のためのコード，EvEmfrp モジュールを起動するための関数呼び出しコードが含まれている．ユーザーは生成されたスケルトンコードに適切な追記を行い，EvEmfrp プログラムを実行する．

5.2.2　EvEmfrp による記述例

EvEmfrp で記述した「ボタンの押下によって LED の発光状態を切り替える」プログラムを図 5.1 に示す．EvEmfrp プログラムは Emfrp と同様にモジュールという単位で記述され，時変値はノードと呼

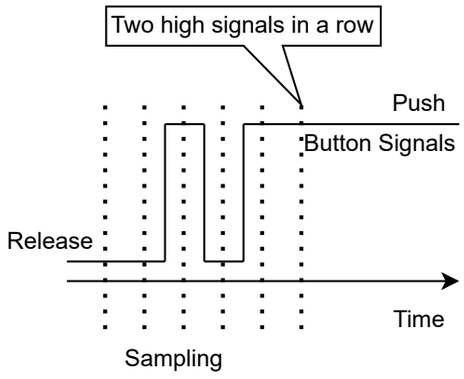

図 5.2　サンプリング方式

ばれるオブジェクトとして表される．ノードには入力ノード，中間ノード，出力ノードがある．プログラム 1 行目はモジュール名，2 行目は入力ノード，5 行目は出力ノードである．入力としてボタンの状態 (押されていたら `True`，離されていたら `False`，初期値 `False`)，出力として LED の状態 (`True`なら点灯，`False`なら消灯) が指定されている．入出力のタイミングはどちらも同様に時刻 0(**位相**) から始まる**周期** 20 ミリ秒が指定されている．この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すると，入力ノード `btn`と出力ノード `led`のためにスケルトンコードが生成される．プログラマはボタンに繋がった GPIO をセンシングするコードや，LED に繋がった GPIO へ値を出力するコードをスケルトンコードへ記述することが想定される．

この例では，ユーザーがボタンを押下する際にチャタリングが起こることを想定している．チャタリングとはボタンのような接点が接触状態になる際に高速に機械的振動を起こす現象である．ボタンが押された直後からある程度の時間はチャタリングにより電気信号が乱れるため，一度しか押されていないボタンが複数回押されたと判定されてしまうことがある．あるいは，ボタンが離された直後にチャタリングが起こり，ボタンの押下が判定されることが起こりうる．これを避けるため，適切にノイズ除去を行ったのちにボタンの押下を判定する必要がある．

今回のプログラムはチャタリングを除去 (信号を安定化) できる十分な時間周期を 20 ミリ秒と設定し，その周期に従ってボタンの状態を取得している (図 5.2)．したがって，プログラム 10 行目にて現在の `btn`と 1 周期前の `btn`(`btn@last[1]`) がどちらも `True`になっていればボタン押下されたとみなし，チャタリングが除去されたボタン信号を表す中間ノード `btn_on`を `True`とする．EvEmfrp では，ノード名に`@last[1]`をつけることで，直前の更新周期での時変値を参照することができる．

出力ノード `led`の更新式は 12–13 行目にて定義されている．初期値は `False`である．条件式 `not( btn_on@last[1]) && btn_on`では，`btn_on`の立ち上がりエッジを検出している．これにより，ボタンが押された直後に LED の状態が反転し，そうでない場合には現在の状態が継続する．

5.2.3　タイミング検査と実行モデル

EvEmfrp プログラムはコンパイルの際に，Emfrp が行うノードについての依存関係の循環検査や型検査に加えて，更新タイミングについての整合性検査が行われる．あるタイミングが指定されているノードの定義式中では，別のタイミングを持ったノードの現在値を参照することはできない．タイミング部分化演算子 (`@>`) やタイミング合成演算子 (`@<`f`>@`)，時変値サンプリング (`@now`) を明示的に使用することで別のタイミングを持つ時変値参照を行うことができる (詳細は [66] を参照)．

時変値定義のタイミング検査と同時に，各ノードに記述されたタイミングの位相や周期の最大公約数を使用してシステム全体の更新周期が算出される．簡単のためすべてのタイミングの位相が 0 だとす

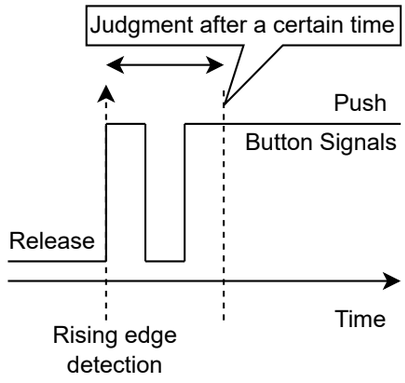

図 5.3　遅延判定方式

ると，システム全体の更新周期は各ノードの更新周期の最大公約数になる．EvEmfrp によって記述されたシステムは，この算出された更新周期あるいは割り込みタイミングごとにスリープモードが解除され，関連したタイミングのノードのみが更新される．同一タイミングを持つノードについては，Emfrp と同様にノードの依存関係による更新順序のスケジューリングが行われ，その順に push 型の時変値更新処理が行われる．すなわち，システム全体の更新周期ごとにその更新時刻に関連したノードのみのイテレーション実行を繰り返すことで，リアクティブな振る舞いを実現している．

5.3　動機：非同期タイミングを起点とする遅延タイミング

EvEmfrp は間欠実行によってシステムの省電力化を図るために時変値の更新タイミングをアノテーションできる機構を持つ．しかし，現状の機能では省電力化が不十分になりうることを前述の例および別の例を提示して説明する．

5.3.1　EvEmfrp のタイミング注釈

EvEmfrp では，モジュールの入力ノードと出力ノードにのみタイミング注釈が許される．タイミング注釈は定数の位相 ϕ と周期 T の 2 つ組で表される周期タイミング (ϕ,T)であるか，シンボルで表される非同期 (割り込み) タイミング$a である．また，タイミング注釈として明示的に記述することはできないが，上記のタイミングの合成を扱うことができる．このような制限は，タイミング整合性の静的な検査を簡便にし，言語ランタイムにも複雑な処理を必要としないための方針である．しかしながら，この制限のため動的に位相や周期が変更されるようなタイミング注釈を表現することができない．結果として「システムの状態に合わせて更新頻度を変更する」,「ある非同期タイミングを起点として，一定時間後に特定の時変値を更新する」,「計算の結果に依存して，特定の時変値のみを更新する」振る舞いを直接的に表現できない．これらの振る舞いを模倣するために，必要以上に頻繁な時変値更新を伴うプログラムが記述されることで消費電力が増加してしまう．

5.3.2　動機となる例

以下の 2 つの例では共にシステムのスリープ状態を長期間維持することができず，省電力効果が不十分になってしまうことを説明する．

ボタンの押下判定

5.2.2 節にて提示したプログラム例について考える．このプログラムの目的は「チャタリングによるノイズを除去しつつボタンの押下を判定し，LED 状態を切り替える」ことである．ボタンの押下は数秒から数分に一度程度だと仮定する．チャタリングによるノイズの除去をソフトウェア的 [*1]に行う方法は主に以下の 2 通り考えられる．

- サンプリング方式：十分な周期でボタンの押下信号をサンプリングしつづけ，信号が安定したことを確認する (図 5.2).
- 遅延判定方式：ボタンが押された瞬間の立ち上がりエッジを検出し，ある程度の時間後にもう一度信号を読み取る (図 5.3).

どちらの方法も信号が安定したのちに押下判定を行なっているため，外乱ノイズ (押していないのに押したと判定されること) に対しては堅牢な方法となっている．

1 つ目の方法 (サンプリング方式) を EvEmfrp によって実装したプログラムが図 5.1 である．プログラム中ではサンプリング周期を 20 ミリ秒と設定し，2 回連続で押下信号を確認できた場合にボタンが押下されたと判定している．すなわち，このプログラムでは 20 ミリ秒ごとにシステムがスリープから復帰し時変値更新計算が行われる．この更新頻度はボタンが押される頻度と比べ明らかに過剰であり，必要以上に電力を消費してしまう．サンプリング周期を長くすればスリープ時間が増え消費電力を削減できるが，システムの応答性が悪化してしまう．

2 つ目の方法 (遅延判定方式) では，最初のボタン押下のタイミングをマイコンの外部割り込み機能によって検出する．外部割り込みが行われるまではシステムはスリープ状態を維持し続けることができるため，1 つ目の方法と比べ消費電力の削減が期待できる．また，一般に外部割り込み機能を有効にしたままスリープモードに移行できるマイコンは多く，現実的な方法だと考えられる．しかしながら，この方法を EvEmfrp で実装することはできない．最初のボタン押下のタイミング (非同期タイミング) を起点にして，ある程度の時間後に時変値更新を行う処理が記述できないためである．当然ながら，最初のボタン押下のタイミングに従って，時変値更新を行うプログラムは EvEmfrp によって簡単に記述可能である．そのようなプログラムでは，ボタンのチャタリングによって，一回のボタン押下に対して複数回の外部割り込みが発生してしまう．

人感センサライト

次に，人感センサによるライティングシステムを例にあげる．このシステムの目的は「人間を検知した場合，10 秒間程度 LED を点灯させ，その後消灯する」ことである．このシステムを EvEmfrp を用いて記述したプログラムを図 5.4 に示す．入力ノード `pirTime`は人感センサが反応した時刻 (ミリ秒)，`timerTime`は位相 10 秒，周期 10 秒ごとの現在時刻 (ミリ秒) を表している．出力ノード `led`は外部接続されている LED ライトの状態を表し，その更新タイミングはプログラムから推論されることを意味している (タイミング注釈_)．システムの振る舞いは 10–12 行目の出力ノード `led`によって定義されている．`@<mergeL>@`はタイミング合成演算子で，右辺左辺のそれぞれの更新タイミングでそれぞれの値に更新され，更新タイミングが同時に重なった場合には左辺の値に更新される (`mergeL`関数) ことを意味する．`@now`演算子は異なる更新タイミングの時変値の値をサンプリングする演算子であり，その時変値の直近に計算された値を参照する．すなわち，出力ノード `led`は人感センサが反応する非同期タイミング$pir と 10 秒ごとの周期タイミングの両方のタイミングで更新される．人感センサが反応したタイミ

[*1] ハードウェア的な除去としては抵抗とコンデンサを使用してローパスフィルタを構築する方法が考えられる．この方法ではノイズ除去後の電気信号の変化が鈍るためデジタル信号入力として扱うためにシュミットトリガが必要になることもある．また，抵抗値や静電容量を適切に調節することが難しい．

```
module PIRLight
in  pirTime : Int '$pir, # clock when motion sensor senses
    timerTime : Int '(10000, 10000) # current clocks
out led : Bool '_ # LED for output

func mergeL(l, r) = l
func ledOn(x) = True

# update at both $pir and (10000,10000)
node init [False] led =
  ledOn(pirTime)
    @<mergeL>@ (timerTime - pirTimer@now < 10000)
```

図 5.4　EvEmfrp による人感センサライトの記述

ングでは，必ず点灯 (`True`) する．周期タイミングでは，現在時刻 (`timerTime`) と直近の人感センサが反応した時刻 (`pirTimer@now`) の差が 10 秒未満ならば点灯させ，そうでないならば消灯させる．

プログラム中の周期タイミング注釈から明らかなように，このシステムは必ず 10 秒おきにスリープモードから復帰する．これは人間がセンセの前を横切る頻度よりも明らかに短い間隔であり，消費電力のさらなる削減が見込めるだろう．また，固定秒数の間隔で現在時刻を取得しているため，正確に 10 秒程度で消灯させることができていない．もちろん周期を短くすればより正確な 10 秒に近づくが，よりスリープ復帰が頻発する．

小規模組込みシステムでは，上記のボタン押下や人感センサライトの例のように，「ある非同期タイミングを起点として，一定時間後に処理を行う」パターンは頻出する．そのようなシステムを記述するに際し現状の EvEmfrp では，このパターンを直接的に記述することができず，また振る舞いを模倣するプログラムでは時変値更新処理が必要以上に行われることが確認できた．

5.4　EvEmfrp/S

本節では，EvEmfrp の持つタイミング注釈記法を刷新し，動的なタイミング切り替えを表現することのできる新たな言語 EvEmfrp/S を提案する．これにより 5.3 節で取り上げた例題を直接的に記述できることを具体例を通して確認し，その有用性を示す．その後，このタイミング注釈が表す振る舞いを実現するためのコンパイル方法および言語ランタイム設計について提示する．

5.4.1　新しいタイミング注釈

EvEmfrp/S では，メインモジュールにて，2 種類の更新タイミング `event`と `periodic`を定義することができる．

`event`で定義される更新タイミングは，割り込み信号のような one-shot な更新タイミングである．`event`によるタイミング定義は次の 2 パターンある．

$$\texttt{event}\ E\ \texttt{=}\ E'\ \texttt{->}\ T$$
$$\texttt{event}\ E\ \texttt{=}\ \texttt{when}\ n$$

E,E' は `event`タイミング名，n はノード名である．T には時間間隔を表す定数を指定することができ，`5 ms`や `10 sec`などと書き 5 ミリ秒や 10 秒であることを指定する．T には 0 は指定できない．一つ目のパターンは，`event`タイミング E' から T 秒後に有効になる遅延タイミングの定義である．もし T 秒以内にもう一度 E' が発生した場合，それまでにカウントされた分は無効になり，E はそのタイミングから T 秒後に有効になる．二つ目のパターンは，`Bool`型のノード n が `True`になった時に有効になるノー

```
main module FanController
in F   : event Interval[10ms],
   tmp : Float at T,
   hmd : Float at H
out fan  : Bool at F

periodic T = init (  0ms, 500ms)
periodic H = init (250ms, 500ms)

newnode di : Float at (T | H) = CalcDI(T, H, tmp, hmd)
node fan = init False | di@last >= th
node th : Float at F =
 75.0 + if fan@last then -0.5 else 0.5
```

図 5.5　EvEmfrp/S による FanControllerモジュール

```
# calc. discomfort index for each timing Et and Eh
module CalcDI
in Et : event, Eh : event,
   t : Float at Et,
   h : Float at Eh
out d : Float at (Et | Eh)

func index(u, v) =
  0.81 * u + 0.01 * v * (0.99 * u - 14.3) + 46.3

node d = init 0
  | Et -> index(t, h@last)
  | Eh -> index(t@last, h)
```

図 5.6　EvEmfrp/S による不快指数算出モジュール

ドの値に依存したタイミングの定義である．ノード n のノード更新後に n が Trueであれば，そのノード更新の直後にタイミング E が有効となったノード更新が発生する．

periodicで定義される更新タイミングは非同期タイミングによって切り替わる周期タイミングを表す．以下の形である．

$$\texttt{periodic}\ P\ \texttt{=}\ \texttt{init}\ (\phi_0, T_0)\ \texttt{|}\ E_1\ \texttt{->}\ (\phi_1, T_1)\ \texttt{|}\ \cdots\ \texttt{|}\ E_n\ \texttt{->}\ (\phi_n, T_n)$$

P は periodicタイミング名，E_i は eventタイミング名，ϕ_i は位相，T_i は周期を表す．システム起動時，periodicタイミング P は，(ϕ_0, T_0) の周期タイミングであることを表す．さらに eventタイミング E_i が発生するたびに，瞬時に (ϕ_i, T_i) の周期タイミングへと切り替わることを表している．タイミング E_i の発生タイミングから数えて ϕ_i 秒後から周期 T_i の周期タイミングが発生する．$\phi_i = 0$ の時には，$when$ による更新タイミングと同様に，連続してノード更新が発生する．また，ϕ_i や T_i に，無限大 INFを指定することで，周期タイミングを停止させることができる．

EvEmfrp/S では，以上の eventタイミングや periodicタイミングをノードに注釈することで，ノードの更新タイミングを制御することができる．

5.4.2　EvEmfrp/S の言語機構

基本的な例題とともに EvEmfrp/S の言語機構を説明する．図 5.5 は温度と湿度から不快指数を算出し，ファンを回転させるモジュール FanControllerである．図 5.6 は FanControllerモジュールからサブモジュールとして利用されている．

モジュール

EvEmfrp/S ではタイミング定義がメインモジュールでのみ行えるため，図 5.5 の 1 行目にて `main`をつけて強調している．EvEmfrp/S ではモジュールはそのモジュールの更新に関連した非同期タイミングをパラメータとして受け取ることができる．図 5.5 の 2 行目と図 5.6 の 3 行目では，関連する非同期タイミングを宣言している．メインモジュールでのみ `Interval[`T`]`によって外部割り込みの受信間隔を設定することができる．これは非同期タイミングが発生してから T の間は同じ非同期タイミングをハンドリングしないことを意味する．つまり，T は同一の非同期タイミングに対するクールタイム (あるいはマスク) と捉えることができる．

図 5.57–8 行目ではメインモジュール内にて `periodic`タイミングの定義を行なっている．この例では，周期は切り替わらないため `init`のみが指定されている．10 行目では，サブモジュールの呼び出し (展開) が行われ，タイミングをサブモジュールへと受け渡している．

ノード定義

図 5.5 3–5，10，12 行目のように，ノード定義時の型に対し `at`をつけることで，そのノードの更新タイミングを指定する．また，12 行目のようにデータ型や更新タイミングが明らかな場合には省略できる．今までの EvEmfrp との主な違いは，図 5.6 6 行目のように合成されたタイミングを記述できることである．さらに，二つ以上の更新タイミングを持つノードに関しては，11–13 行目のように更新タイミングに応じた更新式を記述することができる．この記法は E-FRP [75] に影響された．更新タイミングが唯一で明らかな場合には，図 5.5 の 11 行目のような省略を許す．

EvEmfrp では，更新順序のスケジューリングのためにノードの依存関係に循環がないかどうか静的に検査される．EvEmfrp/S では，更新タイミングごとにノードの依存関係に循環がないかどうか静的に検査する．また，同じ更新タイミングを持たないノード同士の参照は禁止する．つまり，例えば図 5.6 12 行目にて `h`を参照することはできない．ただし，`@last`による参照は可能である．この制限は，ユーザーが更新タイミングについて混乱するのを避けるためである．

5.4.3 EvEmfrp/S による例題の記述

5.4.1 節で提案したタイミング注釈を用いて記述した EvEmfrp/S によるプログラム例を提示する．

遅延判定方式によるボタン押下判定

5.3.2 節にて示した遅延判定方式でボタンの押下判定するプログラムを図 5.7 に示す．タイミング `Ea`(2 行目) はボタンの立ち上がりエッジを検出した際に発生する非同期タイミングである．`Ta`には 10 ミリ秒間のインターバルが設定されているため，チャタリングによる高頻度な割り込みは起こらない．タイミング `Eb`(7 行目) はタイミング `Ea`が発生してから 10ms 後に発生するタイミングを表している．入力ノード `btn_status`(3 行目) はタイミング `Tb`が指定されているため，安定した後のボタン信号をセンシングできる．タイミング `Ec`(10 行目) は，入力ノード `btn_status`が `True`になったときに発生するタイミングである．タイミング `Ec`が指定されている出力ノード `led`はボタンの押下判定が成功したときにのみ更新処理が行われる．以上の振る舞いから，このプログラムではボタン押下を判定するために高頻度のセンシングをする必要がない．そのため，スリープ状態を長期間維持することができ，省電効果を高めることが期待される．

サンプリング方式と遅延判定方式のハイブリッドによるボタン押下判定

EvEmfrp/S が持つ遅延タイミングや切り替え可能な周期タイミングを組み合わせることで，サンプ

```
1  main module BlinkLED
2  in Ea : event Interval[10ms], # leading edge event
3     btn_status : Bool at Eb
4  out led : Bool at Ec
5
6  # 5ms after the edge rises
7  event Eb = Ea -> 10ms
8
9  # call the button is pressed
10 event Ec = when btn_status
11
12 # update the output for LED with timing Ec (invert)
13 node led = init False | not(led@last)
```

図 5.7　EvEmfrp/S による遅延判定方式でのボタン押下判定

リング方式と遅延判定方式を組み合わせた振る舞いをプログラムすることができる．すなわち，待機状態からボタンの立ち上がりを起点にサンプリング動作を開始し，一定時間後に再び待機状態へと戻ることを繰り返す．これを「ハイブリッド方式」と呼ぶ．

図 5.8 にハイブリッド方式でのボタン押下判定を行うモジュール `BlinkLEDHybrid`を示す．このプログラムは，ボタンの立ち上がりを検出すると周期 20ms のサンプリング動作を 5 秒間行なったのち，待機状態に戻ることを繰り返す．

タイミング `Eedge`(2 行目) はボタンの立ち上がりエッジのタイミングである．タイミング `Eend`(6 行目) は `Eedge`の 5 秒後に発生する遅延タイミングである．タイミング `Psample`(7–9 行目) はタイミング `Eedge`からタイミング `Eend`までの間に 20 ミリ秒ごとに発生する周期タイミングである．ノード `btn`(11–13 行目) はタイミング `Eedge`とタイミング `Psample`の発生によって更新される．タイミング `Psample`はタイミング `Eedge`が発生した直後に発生する (`Eedge`からの位相が 0 なので)．そのため，タイミング `Eedge`が発生した時に `btn`を `False`にすることで，その直後のタイミング `Psample`での `btn@last`が `False`にリセットされるようになっている．ノード `btn_on`(15 行目) はタイミング `Psample`に従って，ノード `btn`が 2 回連続で high であるかどうかを検出する (図 5.1 の 9 行目と同様の記述)．これにより，ノード `btn_on`はチャタリングが除去されたのちのボタンの入力信号として扱うことができる．ノード `push_pulse`はノード `btn_on`の立ち上がりタイミングを検出する．検出されたタイミングはタイミング `Eout`(17 行目) として利用される．このプログラムでは，ボタンの押下 (ノイズかもしれない) に反応して，電力を消費するシステムの周期的更新を一定時間行う．従来の EvEmfrp では固定の時変値更新周期であったため，システムの状態に応じて更新頻度を下げ消費電力を抑えることができなかった．提案注釈によって新たに従来の消費電力を抑える手法を利用することができるようになったことを示す例となっている．

人感センサライト

5.3.2 節にて提示した人感センサライトを提案手法によって記述した例を図 5.9 に示す．割り込みタイミング `Ep`(2 行目) は人感センサが反応した瞬間の立ち上がりエッジを表す非同期タイミングである．遅延タイミング `Eq`(8 行目) はタイミング `Ep`から 10 秒後に発生する LED を消灯させるためのタイミングである．ここで，タイミング `Ep`のインターバルは 1 秒であるから，1 秒おきに人感センサが反応する可能性がある．人感センサが反応するたびに `Ep`が発生するため，タイミング `Eq`の発生もそのぶん遅延する．すなわち，センサが反応してから次に反応するまでに 10 秒以上の時間があるときに，`Eq`が発生する．このモジュールは入力ノードを持たない．出力ノードは `led`(4 行目) で，`Ep`が発生した時に `True`(点灯)，`Eq`の時に `False`(消灯) へと更新される．

```
main module BlinkLEDHybrid
in Eedge : event Interval[1000ms],
   btn_status : Bool at Psample
out led : Bool at Eout

event Eend = Eedge -> 5000ms
periodic Psample = init (INF, INF)
           | Eedge -> (0, 20ms)
           | Eend -> (INF, INF)

node btn = init False
         | Eedge   -> False
         | Psample -> btn_status

node btn_on : Bool at Psample = btn@last && btn

node push_pulse : Bool at Psample = not(btn_on@last) && btn_on

event Eout = when push_pulse
node led = init False | not(led@last)
```

図 5.8　EvEmfrp/S によるハイブリッド方式でのボタン押下判定

```
main module PIRLight
in Ep : event Interval[1sec]
        # leading edge of the motion sensor
out led : Bool at (Ep | Eq) # LED for output

# 10sec after Ep
# extended 10 sec for every Ep
event Eq = Ep -> 10sec

# update the output of LED with timing Ep and Eq
node led = init False
  | Ep -> True  # turn on when a person is detected
  | Eq -> False # turn off
```

図 5.9　EvEmfrp/S による人感センサライトの記述 (センサ反応ごとに点灯時間延長)

```
main module PIRLight2
in Ep : event Interval[10sec] # 10 sec!!
out led : Bool at (Ep | Eq)

# about 10sec after Ep
event Eq = Ep -> 9900ms

node led = init False
  | Ep -> True | Eq -> False
```

図 5.10　EvEmfrp/S による人感センサライトの記述 (約 10 秒間点灯，点灯時間延長なし)

人感センサが追加で反応しても LED の点灯時間を延長しない場合のプログラムは図 5.10 のようになる．このプログラムの重要なポイントは人感センサの反応タイミングである `Ep`に対して 10 秒のインターバルが設定されていることである．これにより，一度センサーが反応してから 10 秒間はセンサーによる割り込みがハンドルされない．タイミング `Eq`の遅延時間を 10 秒より短くすることで，`Ep`の発生による `Eq`の遅延を防ぐことできる．

これらのプログラムにおいて図 5.4 とは異なり，10 秒ごとの周期的なシステムのウェイクアップは必

要ない．また，新しいタイミング注釈によって「あるタイミングから 10 秒後」のタイミングを指定できるため，より正確な時間幅で LED を点灯することが可能になっている．

5.4.4 実行モデル

この節では，EvEmfrp/S の実行モデルを説明し，それをマイコン上で実現するためのコンパイル方法の概要を提示する．

マイクロイテレーション

EvEmfrp/S において，メインモジュールで定義されるタイミング (`event`や `periodic`) は，ノードの更新タイミングを表す．EvEmfrp/S では，定義された更新タイミングが発生するごとに，対応したイテレーションを行う．このイテレーションをマイクロイテレーション (micro-iteration) と呼ぶ．マイクロイテレーションは，特定の一つのタイミングと対応する．例えばタイミング `T`に対応したマイクロイテレーションでは，`T`の更新タイミングを持つノードのみを依存関係の順番に評価し，値を更新，直前値とメモリの管理を行う．EvEmfrp/S のプログラムは外部割り込みや時間経過によって発生する更新タイミングに対し，対応したマイクロイテレーションを行う．

タイミングの順序関係

コンパイラによって，メインモジュールで定義したタイミング同士の依存関係に循環がないことが静的に検査される．タイミングは，定義された順序にしたがった全順序関係を持つ．この全順序関係はタイミング同士の依存関係の閉包と見なすことができる．

EvEmfrp/S のイテレーション

2 つ以上の更新タイミングが同時に発生した時，タイミングの全順序に従って対応したマイクロイテレーションが連続的に行われる．また，マイクロイテレーションによって後続のタイミングが発生し，マイクロイテレーションが連鎖的に行われることがある．これは，`when`によるタイミングや切り替え後の位相が 0 に設定された周期タイミング (`periodic`) が存在することから引き起こされる現象である．EvEmfrp/S ランタイムは処理すべき対象がなくなるまで，連続的にマイクロイテレーションを行う．現時点で処理すべきマイクロイテレーションが無くなった場合，システムはスリープ状態へと移行する．スリープからの復帰は時間経過によって発生する更新タイミングや，割り込みによる更新タイミングである．マイクロイテレーションの連鎖的処理およびタイミング待ちのためのスリープ処理の一連の流れを EvEmfrp/S におけるイテレーションと呼ぶ．EvEmfrp/S ランタイムはこのイテレーションを繰り返し行うことで，省電力にリアクティブな振る舞いをし続ける．

C 言語へのコンパイル

EvEmfrp/S におけるイテレーションをマイコン上で実現するためのコンパイル方法の概要について説明する．なお，EvEmfrp/S が対象とする動作環境は，OS が載っていない Arduino や STM32 のようなシングルコアマイコンである．図 5.8 を手動でコンパイルした結果を付録 B のソースコード B.1 に提示した．

EvEmfrp/S ランタイムはウォールクロックと呼ばれる時刻を記録する変数を持つ．ウォールクロックはシステムが開始してからのマイクロイテレーションが行われた時間およびスリープしている時間の累計である．後述するタイマーイベントは，このウォールクロックの時刻を参照して行われる．

メインモジュールにて定義したタイミングは，ランタイム上では 3 種類のイベント (割り込みイベント，タイマーイベント，`when`イベント) として管理される．割り込みイベントは割り込みタイミングと

```c
void miter_T(uint32_t current_time){
  /* Configure settings related to timing T
  If T is interrupt timing, set interrupt enable interval.
  If T is one-shot delayed timing, set the next occurrence time of T to infinity(disable T).
  If T is periodic timing, set the next occurrence time of T using the current period.
  */

  // Update nodes with timing T.
  // Manage @last value and memory.

  // Set up another timing that is affected by timing T.

}
```

図 5.11　タイミング Tのためのマイクロイテレーションの擬似コード (C 言語)

対応する．whenイベントは whenタイミングと対応する．タイマーイベントは，遅延タイミングと周期タイミングの管理に加えて，割り込みタイミングのインターバルを管理する．割り込みタイミングのインターバル管理は，遅延タイミングや周期タイミングによるマイクロイテレーションの代わりに割り込み許可を与えるユーザー定義関数を呼び出す操作としてコンパイルされる．

タイミング Tのマイクロイテレーションは図 5.11 に示す擬似コードの形式で生成される．初めにタイミング Tに関する操作 (周期の設定や one-shot タイミングの無効化など) を行う．その後，タイミング Tに関するノード更新をする．最後にタイミング Tが影響を及ぼす他のタイミングの状態を変更する．

全体のイテレーションは，マイクロイテレーションを繰り返し行なったのちにスリープ状態へと遷移する動作を繰り返すループとしてコンパイルされる．ユーザーは，タイマーの設定関数や入出力関数などのスケルトンコードを補完して，システムを構築する．

5.4.5　タイミング定義の制限

サブモジュール内でのタイミング定義が課題の一つである．現状では，タイミング定義はメインモジュールしか許されていない．サブモジュール内でタイミング定義を行うと，親モジュールから観測できない更新タイミングが定義できる．そのため，サブモジュールの出力ノードを適切に扱えなくなってしまう．この問題を克服することでより，柔軟な更新タイミングの設定が行えると考えている．

5.5　評価

EvEmfrp/S では，リアクティブシステムを記述する際に必ずしも周期的な時変値更新を行う必要はない．この性質による省電力効果を確かめるために，ボタン押下の例を対象として，消費電流を計測し EvEmfrp と比較した．

5.5.1　問題設定

評価実験の対象となるプログラム，プログラムの実行環境，結果の計測方法について説明する．

対象プログラム

5.3.2 節にて提示したボタンの押下判定を例題として評価実験を行なった．EvEmfrp によるプログラムは図 5.1 を使用した．このプログラムはサンプリング方式でボタン押下を判定する．ただし，入力ノード名を input_btnと変更した．このプログラムのための入出力関数 (C 言語) を図 5.12 に示す．入

```
void Input_input_btn(bool &input_btn) {
  GPIO_PinState s = HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_7);
  input_btn = (s == GPIO_PIN_RESET); // active low
  //printf("INPUT FROM SENSORS.\r\n");
}

void Output_led(const bool &led) {
  //printf("OUTPUT TO ACTUATORS.\r\n");
}
```

図 5.12　BlinkLEDモジュール (EvEmfrp) のための入出力関数

```
void Input_btn_status(bool* btn_status){
  GPIO_PinState s = HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_7);
  *btn_status = (s == GPIO_PIN_RESET); // active low
  //printf("INPUT FROM SENSORS.\r\n");
}

void Output_led(bool* led){
  //printf("OUTPUT TO ACTUATORS.\r\n");
}
```

図 5.13　BlinkLEDHybridモジュール (EvEmfrp/S) のための入出力関数

力関数では，STM マイコンのライブラリ関数によって GPIO ポート B の 7 番ピンから入力値を得ている．ボタンはアクティブローで接続されているため，極性を反転させて EvEmfrp モジュールへの入力としている．入力関数と出力関数の末尾では，printf 関数の呼び出しによってシリアル通信 (USART) を行うコードがコメントアウトされている．評価実験にて，入力や出力に対する負荷の違いによる消費電流の変化を測定際に，このコメントアウトを外し負荷として扱った．したがって，入出力の負荷をかけない場合には，入力関数は入力ピンの状態を取得し，出力関数は何も行わないということになる．

EvEmfrp/S によるプログラムは図 5.8 を使用した．このプログラムでは，5.4.3 節で説明したようにハイブリッド方式によるボタン押下判定を行う．前述の EvEmfrp プログラム (サンプリング方式) と比較して，ボタン押下の待機の際に頻繁にウェイクアップが要求されないことから，消費電力の削減効果が期待される．このプログラムのための入出力関数を図 5.13 に示す．EvEmfrp プログラムのものとほぼ同様である．

実行環境

プログラムの実行には，Emfrp^{HT} の評価実験と同様に ST マイクロ社のマイコンボード NUCLEO-F401RE を使用した．今回の評価実験では，動作クロックを 84MHz に設定した．Emfrp^{HT} の評価実験と同様に，プログラムのコンパイルおよびリンクには STM32CubeIDE(Version 1.14.0) に付属するコンパイラ (arm-none-eabi-gcc) を使用した．最適化オプションは`-Os`によるサイズ最適化を指定した．

今回の評価実験では，ハードウェアスリープを活用する．STM32F4 マイコンには，今回の評価対象プログラムに適したハードウェアスリープのモードが 2 種類用意されている．1 つ目は SLEEP モードである．このモードは MCU 中の CPU へのクロックのみを停止し，周辺ペリフェラルへのクロック供給は継続する．そのため，省電力効果は限られるが，通常タイマーによる割り込みや UART モジュールからの受信割り込みなどでスリープモードを解除 (ウェイクアップ) することができる．2 つ目は STOP モードである．このモードは MCU 中の CPU へのクロックおよび周辺ペリフェラルへのクロック供給を停止する．このモードでは，限られた一部のペリフェラルのみが作動を続けるため，高い省電力効果を受けることができる．しかし，スリープモードの解除には，外部入力端子やリアルタイム

クロックの割り込みによるウェイクアップに限られる．今回の評価実験では，ボタン押下による外部端子からの割り込みを活用するため，SLEEP モードと STOP モードのどちらのハードウェアスリープも使用可能である．

EvEmfrp はコンパイラによって FreeRTOS の使用を前提とする C 言語コードへと変換される．評価実験では，STM32CubeIDE(Version 1.14.0) に付属する FreeRTOS(CMSIS_V2) を使用した．評価対象の EvEmfrp プログラムは既存の EvEmfrp コンパイラによって C 言語コードへと変換した．これと C 言語による入出力関数記述，STM32F401RE のための周辺コード，FreeRTOS をリンクし，実験のためのバイナリファイルを得た．また，FreeRTOS はヘッダーファイル `FreeRTOSConfig.h`にて適切なマクロを設定することで，機能の取捨選択を行うことができる．今回の実験では，アイドル時にハードウェアスリープを活用するために，`configUSE_TICKLESS_IDLE`マクロを有効にした．FreeRTOS では，SLEEP モードによるハードウェアスリープは可能であったが，STOP モードによるスリープでは正常に動作しなかった．そこで今回の評価実験では，SLEEP モードによるハードウェアスリープのみを対象に測定を行った．

EvEmfrp/S は，FreeRTOS を前提としない言語ランタイムを持つ．その代わりに，タイマや割り込みなどプラットフォームに依存する記述はユーザーが行う．評価実験に際し，タイマ 2 を 10kHz のカウントアップタイマとして設定して使用した．また，ボタンの押下判定のために GPIO ポート B の 7 番ピンに対して外部端子割り込みの設定を行なった．EvEmfrp/S 言語のランタイム上では，次にウェイクアップすべき時刻が設定されている場合のスリープ移行 (スリープ移行 A) と割り込み待ちによるスリープ移行 (スリープ移行 B) が区別されている．ユーザーはこれら 2 種類のスリープ移行をカスタマイズして使用することができる．そこで，今回の評価実験では，(スリープ移行 A) については SLEEP モードへの遷移をおこなった．(スリープ移行 B) については，SLEEP モードと STOP モードへの移行のそれぞれの場合について計測をおこなった．評価対象の EvEmfrp/S プログラムは著者が C 言語コードに手動で変換した．これと C 言語による入出力関数記述，EvEmfrp/S 言語ランタイムのためのプラットフォーム依存記述 (C 言語)，STM32F401RE のための周辺コードをリンクし，バイナリファイルを得た．

計測方法

電流の計測には，Nordic Semiconductor 社の Power Profiler Kit2(PPK2) と付属のアプリケーションを使用した．NUCLEO ボード上のジャンパーピン JP6 と PPK2 を接続することで，MCU に供給される電流を計測することができる．評価実験では，MCU の消費電流のみを対象に計測を行なった．また，NUCLEO ボード上のボタンの電源は MCU と共有されているため，外部の別電源およびボタンを接続して計測を行なった．

5.5.2 計測結果と考察

評価対象に対し，スリープモードの設定や入出力関数内の負荷を変更して消費電流量の変化を計測した．以下ではそれぞれの状況に対する計測の結果を提示し考察を述べる．

測定結果の概要

測定によって得られた電流波形を示す．それぞれのグラフの縦軸は 0 20 ミリアンペア，横軸のウィンドウ幅は 10 秒である．

まず EvEmfrp によるプログラムについて 2 つ提示する．図 5.14 は，入出力負荷なしの `BlinkLED`モジュールの測定結果の一部分である．図 5.15 は，入出力負荷ありのものである．これらの図はともにボタン待機状態のものであるが，ボタン押下による消費電流量の変化は見られなかった．どちらの結果

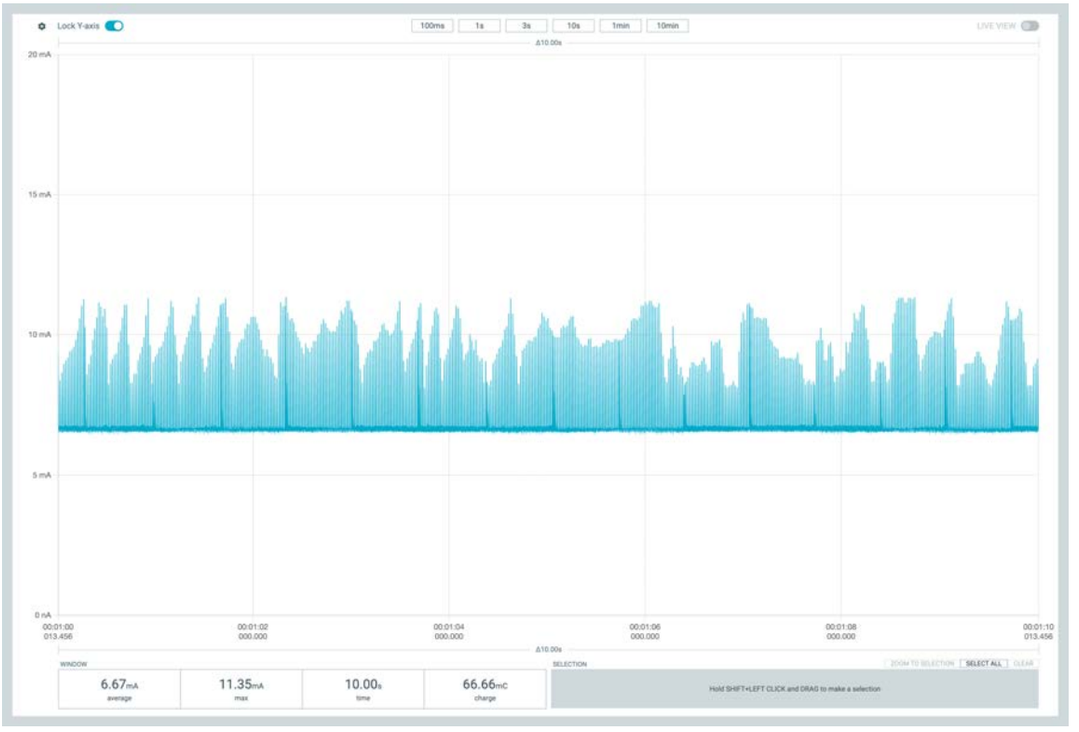

図 5.14　`BlinkLED`モジュール (EvEmfrp，SLEEP，負荷なし) の消費電流

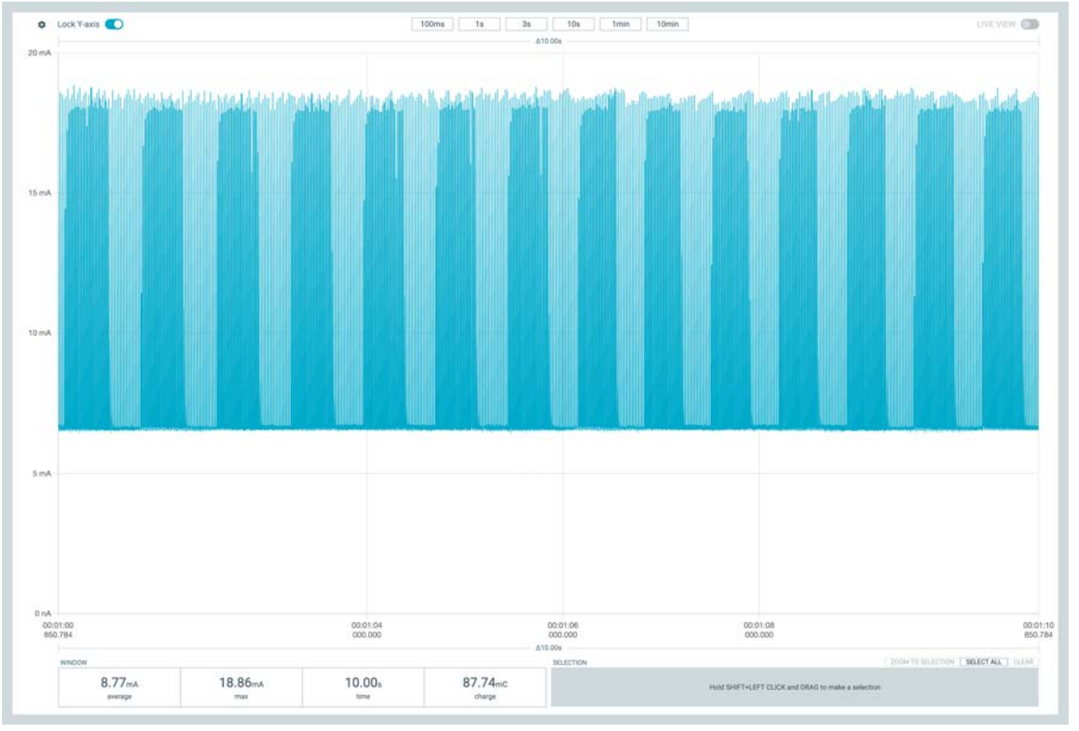

図 5.15　`BlinkLED`モジュール (EvEmfrp，SLEEP，負荷あり) の消費電流

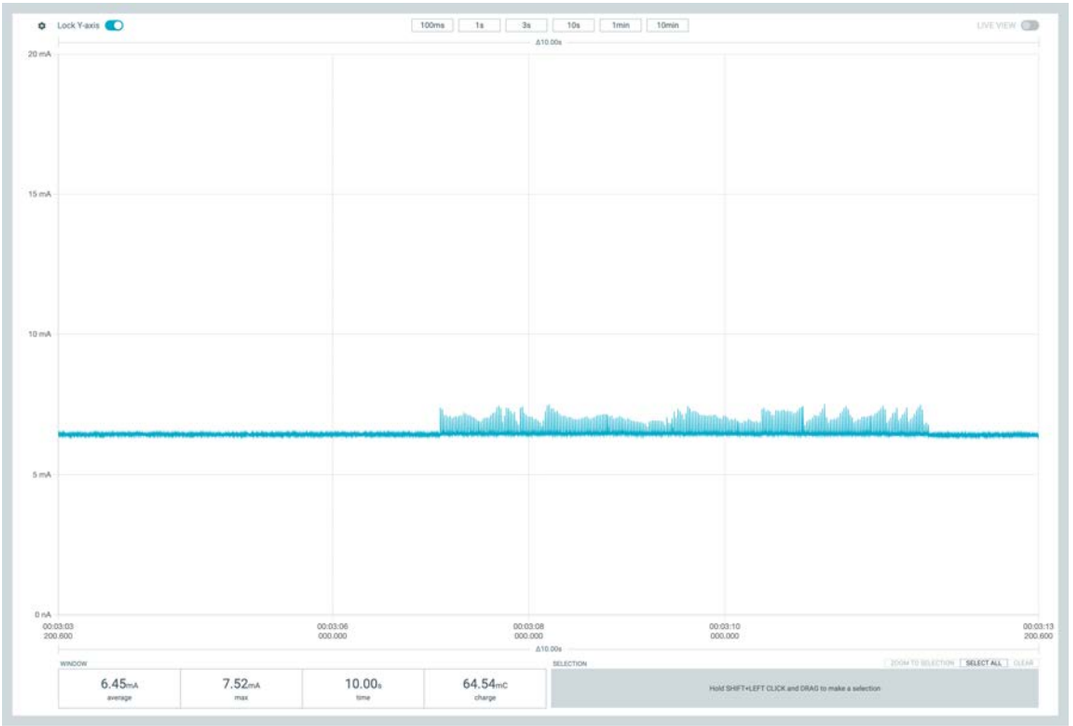

図 5.16　`BlinkLEDHybrid`モジュール (EvEmfrp/S，SLEEP，負荷なし) の消費電流

からも，システムは周期的に動作 (サンプリング方式) していることが確認できる．また，図 5.15 に注目すると，EvEmfrp では高頻度に入力関数や出力関数が呼び出されていることが確認できた．

次に EvEmfrp/S によるプログラムについて 4 つ提示する．これらの計測では，途中にボタン入力を行なった．そのため，5 秒間のボタンサンプリング動作の様子を確認することができる．図 5.16 は，SLEEP モード待機，入出力負荷なしの `BlinkLEDHybrid`モジュールの測定結果の一部分である．図 5.17 は，SLEEP モード待機，入出力負荷ありのものである．図 5.18 は，STOP モード待機，入出力負荷なしのものである．図 5.19 は，STOP モード待機，入出力負荷ありのものである．これらのグラフから，ボタン押下に応じてサンプリング動作が開始し，約 5 秒ほどで待機状態へと戻る振る舞い (ハイブリッド方式) が確認できた．図 5.17 や図 5.19 に注目すると，ボタン押下待機状態では入力関数や出力関数は呼び出されていないことが確認できた．

以上のグラフに共通することとして，入出力に負荷がある場合には最大電流が上昇することが確認できた．また，図 5.14 から，FreeRTOS によるウェイクアップが高頻度に起きていることが確認できる．一方で，ウェイクアップしている時間は短いため，平均電流量には大きな影響はない (後節にて数値的に確認する)．図 5.16 と図 5.18 を比較することで，SLEEP モードと STOP モードの消費電流の差を確認することができる．ボタン押下待機の際，SLEEP モードでは約 6 ミリアンペアに対して，STOP モードでは 0 に近い値になっている．続く節では，以上の観察を数値的に確認する．

待機時の電流

上記のそれぞれの評価対象について，ボタン押下待機状態での 3 分間の電流計測 (10kHz でのサンプリング，合計 1800000 サンプル) を行なった．結果を表 5.1 に示す．まず，同じ記述言語のプログラム同士を比較する．入出力に負荷がある場合には，EvEmfrp プログラム同士を比較すると標本平均で 2.1mA ほどの差を確認することができた．EvEmfrp/S プログラム同士の比較では，スリープモードが同じならば，ほとんど差はなかった．これは，EvEmfrp/S プログラムの待機状態には入出力関数は呼び出されないためである．また，EvEmfrp/S プログラムにおいて，SLEEP モードと STOP モードで

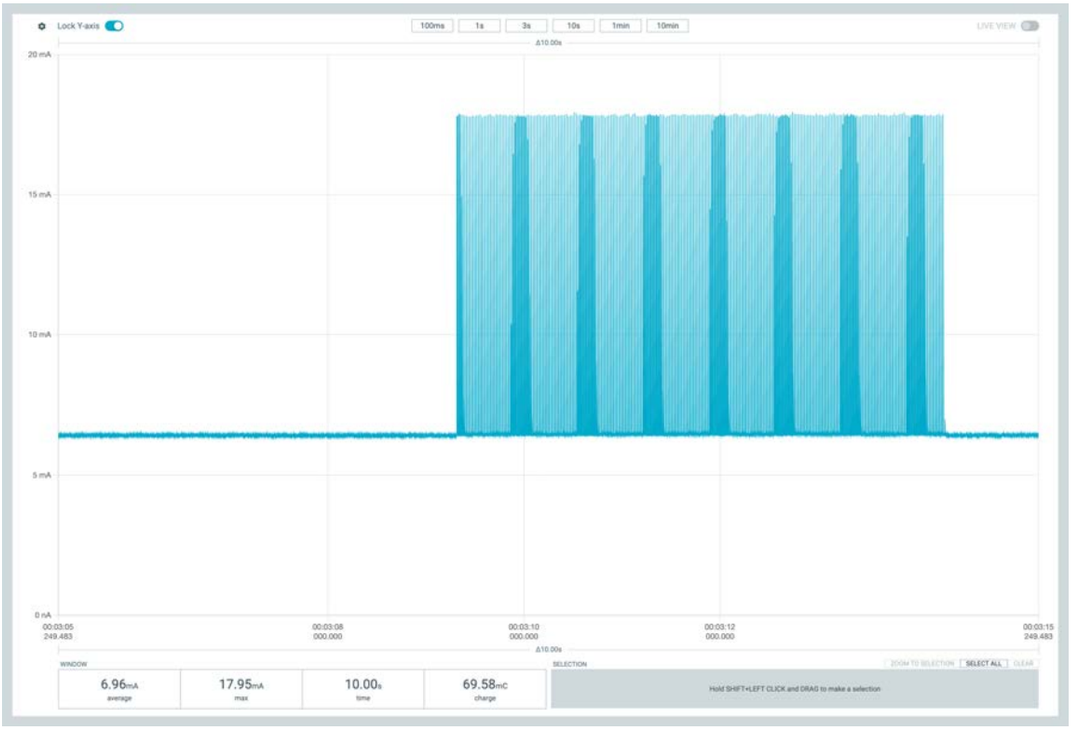

図 5.17　`BlinkLEDHybrid`モジュール (EvEmfrp/S，SLEEP，負荷あり) の消費電流

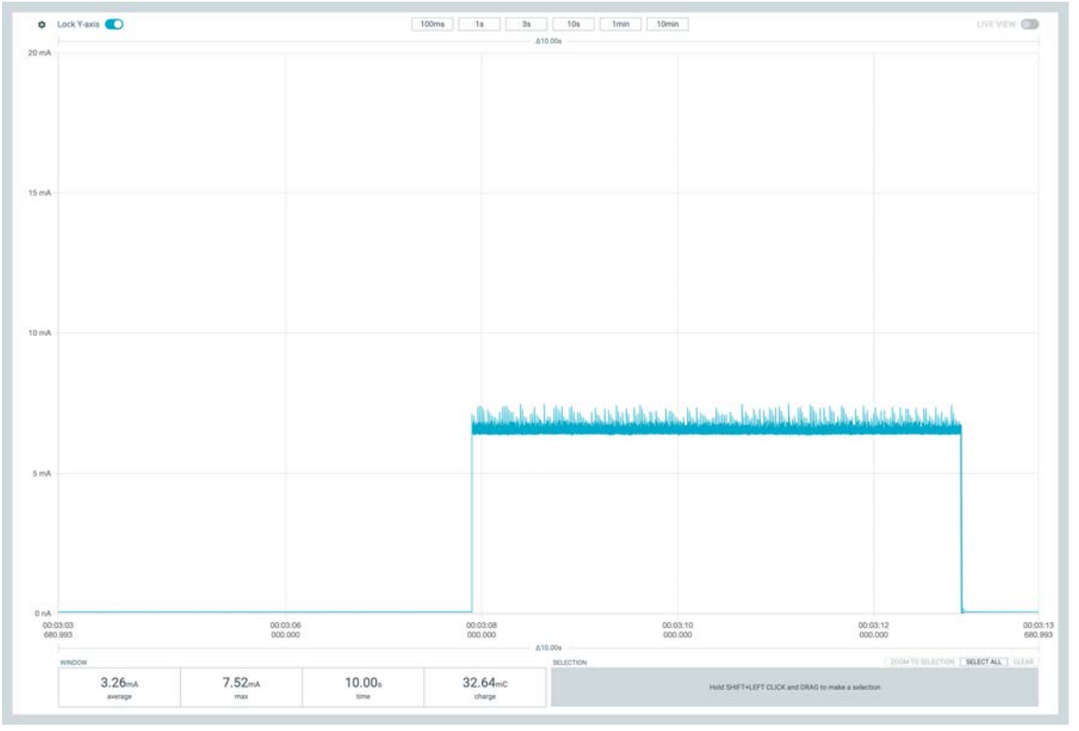

図 5.18　`BlinkLEDHybrid`モジュール (EvEmfrp/S，STOP，負荷なし) の消費電流

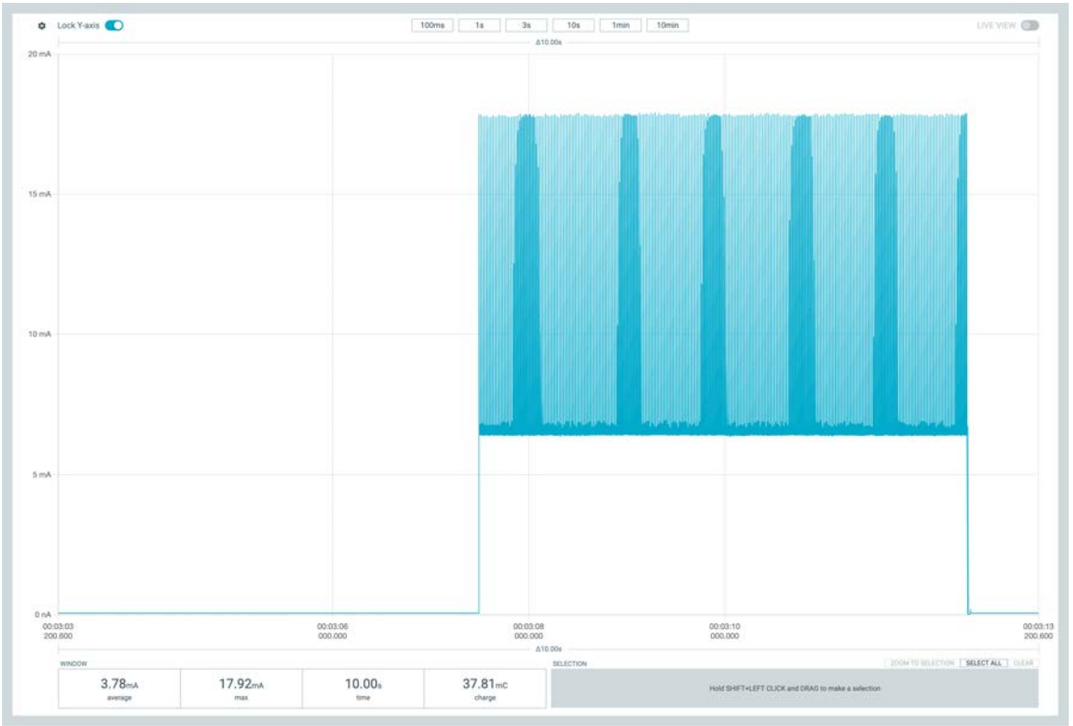

図 5.19　`BlinkLEDHybrid`モジュール (EvEmfrp/S，STOP，負荷あり) の消費電流

表 5.1　待機状態の消費電流

言語 *1	負荷 *2	最小	最大	標本平均 *3	不偏分散	信頼区間 *4	総電荷量
			スリープモード：SLEEP				
		[mA]	[mA]	[mA]		[mA]	[C]
E	なし	6.483	11.368	6.659	0.066	[6.658, 6.660]	1.20
E	あり	6.472	18.938	8.770	18.961	[8.737, 8.804]	1.58
S	なし	6.299	6.530	6.434	0.001	[6.434, 6.434]	1.16
S	あり	0.646	6.550	6.436	0.001	[6.436, 6.436]	1.16
			スリープモード：STOP				
		[uA]	[uA]	[uA]		[uA]	[mC]
S	なし	55.847	70.874	63.809	2.238	[63.802, 63.817]	11.48
S	あり	56.773	71.027	63.620	2.208	[63.612, 63.627]	11.44

*1 プログラム記述に使用した言語 (E:EvEmfrp，S:EvEmfrp/S)
*2 入出力負荷 (入出力関数内で UART 通信 (送信) を行う)
*3 3 分間計測 (10kHz でのサンプリング，合計 1800000 サンプル)
*4 t 分布における平均値の 95% 信頼区間

は明らかな消費電流の差が見られた．

次に異なる記述言語のプログラムを比較する．入出力の負荷がない場合，SLEEP モードによる待機では，EvEmfrp と EvEmfrp/S の消費電流にはほとんど差がなかった．EvEmfrp プログラムでは少なくとも 20 ミリ秒の周期ごとにウェイクアップが起こる．入出力関数への負荷がない場合には，ウェイクアップしている時間が短いため電流消費が抑えられていると考えられる．一方で，入出力関数に負荷がある場合には入力関数や出力関数の実行時間が延長されるため，ウェイクアップしている時間も増大する．そのため，EvEmfrp による記述では消費電流が増加している．

表 5.2　セクションサイズの比較

モジュール名 (記述言語)	text [byte]	data [byte]	bss [byte]	合計 [byte]
`BlinkLED`(EvEmfrp)	17360	116	21100	38576
`BlinkLEDHybrid`(EvEmfrp/S)	10128	112	2144	12384

以上より，ウェイクアップ時間が十分に短ければ EvEmfrp による周期的な振る舞いでも電力消費を抑えられることが確認できた．しかし，シリアル通信による出力を行うなど，ある程度の時間を要する場合には，EvEmfrp は不必要なウェイクアップによって消費電力を増加させてしまう．EvEmfrp/S では，どちらの場合でも不必要な時変値更新処理を抑制し電力消費を削減することが可能である．

空間オーバーヘッド

入出力負荷なし，SLEEP モードによるスリープを行う `BlinkLED`モジュール (EvEmfrp) と`BlinkLEDHybrid`モジュール (EvEmfrp/S) について，バイナリファイルのセクションサイズを比較した．結果を表 5.2 に示す．表から bss 領域の大きさが顕著に異なることがわかる．これは，EvEmfrp がイテレーション実行のために FreeRTOS 上のタスクを生成することに起因すると考えられる．現状の EvEmfrp では，生成されるタスクのスタックサイズが固定されているため，プログラムによっては大きすぎる領域を確保している場合がある．EvEmfrp/S では，RTOS に依存しないことで，軽量なランタイム構成になっていることが確認できた．

5.6　関連研究

本研究では，動的に切り替わる時変値更新タイミングを表現するための注釈記法，およびその振る舞いを実現するための言語ランタイム設計を検討した．時間に関する注釈を用いて，システムスリープのタイミングを記述する言語機能を持つドメイン特化言語として mTask [37, 44] がある．mTask は，純粋関数型言語 Clean [7, 8] をホスト言語として IoT アプリケーション向けコードの生成を行うための EDSL である．Crooijmans ら [11] は mTask に対し，**リフレッシュレート (Refresh Rate)** と呼ばれるタスク実行の時間間隔を扱う仕組みを導入した．このリフレッシュレートによってタスクの実行がスケジューリングされ，システムがスリープ状態へ遷移可能かどうか判定される．この際，相対時間によって記述された時間間隔を，システム起動時からの絶対時刻に変換してタスク実行を管理している．この手法から，本研究の提案手法におけるランタイム設計の着想を得た．

Céu [56] は Esterel [6] をベースにした Structured Synchronous Reactive Programming 言語である．リソースが制限された組込み環境を対象とした命令的言語で，イベント発行 (`emit`)，イベント待ち (`await`)，構造的並列処理 (`par/or`) などを組み合わせてプログラムを記述する．Céu でも，システム全体がイベント待ちになった際に，スリープモードに移行することで省電力化を図る機構が提案されている [57]．Céu では言語内に外部割り込み設定に関する特別な記法が用意されており，より詳細な情報を使ってスリープモードの選択を行うことができる．我々の研究でのスリープもシステム全体が割り込み待ちになる際に行われる点で類似している．一方で，基礎とするプログラミングスタイル (FRP か Esterel スタイルか) が異なるため，時間に関する記法などは異なる点が多い．

Rhine [5] は，**Monadic Stream Function** を基礎とする Haskell 製の FRP ライブラリである．型レベルクロックを注釈できる機能を持ち，異なる更新周期をもつシグナル関数を組み合わせてシステム全体を記述することができる．Rhine では，異なる更新周期の取り扱い方 (バッファリングや最新のみ保持など) を用途に合わせてユーザーが記述することができる．Rhine は異なる更新周期を組み合わ

せる際の不整合を静的に検出することを目的としている点が本研究と異なる．本研究では，更新周期の動的な切り替えについて注目し，静的な周期更新では実現できない振る舞いの記述を目指した．

組込みあるいはIoTシステム向けFRP言語処理系としてHailstorm [58], Juniper [25], ReactiFi [67]が挙げられる．Hailstormは Arrowized FRP [41, 47] の構文や計算モデルに影響を受けたIoTアプリケーション向け関数型言語である．言語構文は異なるものの，その実行モデルはEmfrpのものとほぼ同様である．すなわち，ある種のポーリングによって入力を待ち，時変値更新や結果の出力を繰り返す．そのため，Emfrpと同様の電力消費に関する問題を持つ．

Juniperはマイコンボード Arduino [3] 向けのFRP言語である．Juniperでは，時変値オブジェクトに対し，`map`関数や`foldp`関数を使用して動的にデータフローグラフを構築する．しかしながら構築されたフローグラフ上の値の伝搬は最外ループによって行われ，システムをスリープに移行することによる省電力化などは行われていない．

ReactiFiはモバイルデバイス上のWiFiファームウェアを記述するために設計されたリアクティブプログラミング言語である．ReactiFiでは，分散リアクティブプログラミングライブラリ REScala [55]に類似した記法でプログラムを記述する．データの入力 (`Source`) から出力 (あるいは副作用の実行，`observe`) までを`map`関数や`fold`関数のメソッドチェーンの形で記述する．`Source(Timer(10ms))`のような記述によって周期タイミングによるデータ入力を行うことが可能である．しかし，動的にタイミングを切り替える場合には，出力における副作用としてReactiFiの外側でタイミングを再設定しなければならない．本研究で提案する注釈記法およびランタイムでは，タイミングの動的な切り替えがソースコード上に明示されるため，より可読性のあるプログラム記述が可能である．

実時間に関するプログラム注釈を利用して，処理のスケジューリングを行う処理系としてSynchron VM/API [59] が挙げられる．Synchron VMは組込みシステム向けの関数型言語処理系である．ユーザーは同期的なメッセージパッシング方式の通信機構を備えた関数型言語 (Concurrent ML [52]に類似) およびSynchron APIを利用してプログラムを記述する．Synchron APIは，TinyTimber kernel(RTOSの一種) [40] に影響を受けたタイミング指定を伴った同期通信関数を提供している．ユーザーは同期的通信の (相対) 開始時間と締め切り時間を注釈しながら，ソフトリアルタイムなアプリケーション記述を行うことが可能になっている．この注釈は非同期実行のスケジューリングに利用される．本研究のタイミング注釈もシステムのスリープ状態への移行をスケジュールするために使用される点で類似している．Synchron APIでは，時間注釈は関数の引数として表現されているため動的に変更されることがある．一方で，本研究では言語ランタイムを簡素にするため静的な周期の選択のみを許している．

5.7 むすび

本研究では，Emfrpの省電力化を意図して開発された後続言語EvEmfrpのもつタイミング注釈が，表現力の不足によりある種のプログラムにおいて不必要な時変値更新が行われることを指摘した．これを解決するために，動的なタイミング切り替え機構をもつ新たなタイミング注釈を提示，EvEmfrp後続言語としてEvEmfrp/Sを提案し，そのコンパイル方針の検討を行なった．

新たなタイミング注釈を使用することで，「システムの状態に合わせて更新頻度を変更する」プログラムや「ある非同期タイミングを起点として，一定時間後に特定の時変値を更新する」プログラムの直接的な記述が可能になった．これによりEvEmfrpでは実装できなかった低消費電力化方法を，FRP言語内で実装可能になった．またシステムスリープの長期化を実現できることから消費電力の削減の機会が増加した．これにより，例えばバッテリー駆動デバイスでは，今までよりも長期的にデバイスが運用できることが示唆され，安定したシステム構築への寄与が期待できる．

第 6 章

結論

6.1 まとめ

本研究では，不確実な環境のもとでも安定に動作する組込みシステムの開発を支援するプログラミング言語の構築を目的として，計算リソースを適切に検査，制限，制御することのできる (リソースアウェアな) リアクティブプログラミング言語の提案を行なった．これまでに提示した計算リソースの管理について以下にまとめる．

- 第 3 章では，メモリリソースの使用を適切に検査と制限する機構を導入することで，Emfrp の再帰的定義に関する制約が緩和できることを示した ($\text{Emfrp}^{\text{BCT}}$)．再帰データ型を扱いつつもメモリ不足による実行時エラーが起こらないことを保証した．これによりメモリリソースについての安定性が保証されたシステムの記述に寄与することができる．
- 第 4 章では，CPU リソースを適切に分割と制御することができる機構を導入することで，Emfrp のような実行モデルを持つ FRP 言語において，リアクティブな振る舞いと Heavy-task が協調動作できることを示した (Emfrp^{HT})．これにより，システムの応答性を犠牲にせずに計算時間のかかる処理を行うことができる．
- 第 5 章では，間欠実行の間隔を柔軟に選択できる機構を導入することで，連続的なシステムスリープ状態の維持や不必要なセンシングの抑制が可能になり，電力リソースの削減機会が増やせることを示した (EvEmfrp/S)．これにより，従来の省電力化技法の適用が容易になった．

これまでに提案したリソース管理手法によって，今まで FRP 言語の利用に適さなかった環境や状況でも FRP 言語の使用によるデータフローに注目したプログラム記述が可能となる．例えば，長期使用が想定される電池駆動デバイスの開発を考える．このようなデバイスは，CPS 中で利用されるエッジデバイスとしても典型的なものである．Emfrp や Juniper など既存の組込み向け FRP 言語の使用する場合には，それらの言語の機能不足から間欠実行等の消費電力を抑える手法を実装することが難しい，あるいは無理なワークアラウンドによる実装が行われてきた．本研究の提案言語 EvEmfrp/S ではハードウェアスリープを活用するために十分な言語機能を備えている．そのため，プログラマは FRP によるデータフロー記述を行いながらシステムの消費電力を抑える手法を実装することができる．

本研究では，エッデジバイスの計算主体となる計算機にまつわるリソース (メモリ，CPU，電力) を適切に扱う機構をもつリアクティブプログラミング言語を提案した．提案言語の有用性は具体例を通して確認した．本研究にて提案した FRP 言語は，さまざまな環境下で安定動作するエッジデバイス構築のための記述言語の選択肢を広げるものであると結論づける．

6.2 本研究の位置付け

これまでに提案されてきたリアクティブ言語やライブラリ，あるいはデータフロー言語と本研究で提案した Emfrp$^{\mathrm{BCT}}$，Emfrp$^{\mathrm{HT}}$，EvEmfrp/S の機能面での比較を表 6.1 にまとめた．対象の機能をサポートしている場合には ✓，サポートしていない場合には × を表示した．機能の一部をサポートしていたり，類似した機能を持つ場合には △ を，不明な場合には – を表示した．より詳細な比較は 3.6 節，4.5 節，5.6 節にておこなった．

本研究では Emfrp 言語を基礎として再帰データ型への対応，RTHP への対応，より省電力な間欠実行への対応を進め，それぞれの機能について拡張言語を提案した．提案した言語では，序論で述べた既存の組み込みシステム向け FRP 言語の問題点を緩和することができた．

提案言語 EvEmfrp/S では，非同期タイミングを起点に周期が変化するタイミング定義を行うことができる．これをオートマトンのような状態遷移機構によって扱うことで，現状の提案手法よりも簡潔かつ柔軟に記述できるのではないかと考える．リアクティブプログラミング言語上でオートマトンに類似した状態管理機構を持つ言語は表中の Lustre，XStorm の他に Zélus [80] などがある．

表 6.1　リアクティブ (データフロー) 言語やライブラリにおける提案言語の位置付け

言語	記述 [*1]	対象	メモリ量決定可能	再帰データ型	RTHPへの対応	間欠実行	状態管理
Emfrp [60]	F	組込み	✓	×	×	×	×
EvEmfrp [66]	F	組込み	✓	×	×	✓	×
XStorm [46]	F	組込み	✓	×	×	×	✓
Distributed XFRP [64]	F	分散	×	✓	×	×	×
Juniper [25]	P	組込み	×	△	×	×	×
Hailstorm [58]	F	組込み	×	✓	×	×	×
ReactiFi [67]	F	組込み [*2]	✓	×	△	×	×
Lustre [20, 45]	F	組込み	✓	×	×	×	✓
Hume [22]	F	組込み	✓	✓	-	-	×
Stella [72]	F/P	分散	×	✓	✓	×	×
Elm($\leq$ 0.16) [12]	F	GUI(ブラウザ)	×	✓	△	×	×
Céu [56]	P	汎用/組込み	✓	×	-	✓	×
mTask [37, 44]	F	分散	×	✓	-	✓	×
Lingua Franca [43]	F/P	分散	×	△	✓	△	×
Rhine [5]	F	汎用 [*3]	×	✓	✓	△	×
$\text{Emfrp}^{\text{BCT}}$	F	組込み	✓	✓	×	×	×
Emfrp^{HT}	F	組込み	✓	×	✓	×	×
EvEmfrp/S	F	組込み	✓	×	×	✓	×

[*1] F: 宣言的，P: 手続き的
[*2] プログラム可能な WiFi チップのファームウェア
[*3] Haskell 製ライブラリ

6.3 リソース管理手法の統合にむけて

これまでに提案した 3 種類の言語を統合するならば，EvEmfrp/S を基礎として，Bounded-Construction Types を導入したのち，Heavy-task への対応を進めるべきである．EvEmfrp/S は FRP 言語の根幹となる時変値更新処理部分をポーリング方式から間欠実行方式に変更しているため，これを基盤とするのが良い．Bounded-Construction Types の導入については，ガベージコレクション処理に拡張が必要なものの，比較的容易な導入が可能だろう．

Heavy-task への対応は間欠実行方式による時変値更新と衝突しないように導入を進める必要がある．Heavy-task の開始や終了を非同期タイミングとして捉えることで，EvEmfrp/S の枠組みに導入できると考えられる．しかし，ノードの更新は短時間で終了するという仮定のもとで設計された EvEmfrp/S のランタイムシステムでは，時間のかかる計算である Heavy-task を適切に扱うことは難しい．タスクの計算時間を考慮に入れたより高度なスケジューリング機構が必要になると予想する．また，処理時間を事前に予測することができない IO 待ちを含む Heavy-task への対応は今後の課題である．

6.4 今後の展望

これまでに提案した 3 つの言語の課題点と今後の展望を以下に挙げる．

- $\text{Emfrp}^{\text{BCT}}$ では，サイズ制限のついた再帰データ型によって，静的に実行時使用メモリ量を決定しつつ，従来の関数プログラミングで利用されているプログラム記述が可能になることを確かめた．相互再帰的な型定義やサイズ多相な型定義が今後の課題である．また，現状のメモリ量決定アルゴリズムは木構造の使用の際に，容易に組合せ爆発を起こし計算時間が増大してしまう．これを解決するアルゴリズムの構築も今後の課題である．
- Emfrp^{HT} で対象とした Heavy-task は時間がかかる計算処理であった．エッジデバイスを構成する要素であるネットワークデバイスやセンサーデバイスの IO 待ち (通信待ち) による Heavy-task には現状の提案機構では対応ができていない．IO 待ちを適切に扱うリソース抽象化の構築が今後の課題である．
- EvEmfrp/S では計算主体 (CPU) のスリープを対象とした．エッジデバイスにはネットワークデバイスやセンサーデバイスのような周辺機器が接続されている．より省電力なエッジデバイス構築のためにはそれら周辺機器のスリープ状態を積極的に活用する必要がある．計算主体と周辺機器のスリープを統合して扱うことのできる機構の開発が今後の課題である．

本研究での提案は，個々の計算機 (MCU や CPU) のリソースについて扱うものであった．これにより，単一の計算機上で起こる FRP 言語の問題点を解決してきた．序論で述べた CPS では，エッジデバイスはセンサーやネットワークにつながることが前提となる．安定動作するエッジデバイス構築のためには，主体となる計算機の管理だけでなくその周辺デバイスのリソース管理や協調動作が必要不可欠となる．省電力化のために周辺機器と協調動作する仕組みやエッジデバイスの通信を管理する仕組みとリアクティブプログラミングを統合することで，高い抽象度の記述による詳細なシステム構築が実現されることを期待する．

謝辞

本論文および本研究を進めるに過程で多くの方のご指導，ご支援を賜りました．心から感謝いたします．

渡部卓雄先生には指導教員として学部，修士，博士の 6 年間に渡ってご指導いただきました．心から感謝いたします．先生のご指導，ご支援なしには本研究の遂行はなし得ませんでした．

森口草介先生には所属研究室助教として修士，博士の 4 年半に渡り研究に対する議論，助言，示唆をいただきました．感謝いたします．

Xavier Défago 先生，権藤克彦先生，林晋平先生，西崎真也先生には本論文の審査員をしていただきました．審査にあたって貴重なご意見をいただきました．感謝いたします．

荒堀喜貴先生，田村康将先生には西 8 号館 E 棟 8 階での学生生活を送る上で大変お世話になりました．たくさんの会話や議論は刺激的なものでした．感謝いたします．

秘書の高野まゆ子氏には，予算執行や書類提出などで非常にお世話になりました．感謝いたします．

所属研究室の先輩たち，同輩たち，後輩たちに感謝します．楽しくて刺激的な研究生活を送ることができました．特に瀧本哲史氏，鈴木豪氏には博士研究の一部の実装や計測の際に大きな助力をいただきました．感謝いたします．

本研究の一部は「次世代研究者挑戦的研究プログラム　殻を破るぞ！越境型理工系博士人材育成プロジェクト (東京工業大学 SPRING スカラシップ研究学生)」および「東京工業大学つばめ博士学生奨学金」による支援のもと行われました．感謝いたします．

学士課程から博士課程までの学生生活をともに過ごした D 進同盟のメンバーである久米啓太氏，徳田俊平氏，谷口晃大氏に感謝いたします．研究生活を送る上で精神面での支えとなりました．

最後に，今までの学生生活をあらゆる面で支えていただいた両親に感謝いたします．

参考文献

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

[2] Heinrich Apfelmus. Reactive Banana. `https://wiki.haskell.org/Reactive-banana`, Jan. 2016.

[3] Arduino. Arduino (`https://www.arduino.cc/`). `https://www.arduino.cc/`, Accessed Mar. 2023.

[4] 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.

[5] Manuel Bärenz and Ivan Perez. Rhine: FRP with type-level clocks. *SIGPLAN Not.*, Vol. 53, No. 7, p. 145–157, sep 2018.

[6] F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1293–1304, 1991.

[7] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean — a language for functional graph rewriting. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pp. 364–384, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[8] Clean team. Clean 3.0 language report (`https://cloogle.org/doc/`), Accessed Mar. 2023.

[9] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in lustre. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, p. 197–206, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Oracle Corporation. Future (Java platform se 8). `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html`, Accessed Sep. 2022.

[11] Sjoerd Crooijmans, Mart Lubbers, and Pieter Koopman. Reducing the power consumption of iot with task-oriented programming. In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming*, pp. 80–99, Cham, 2022. Springer International Publishing.

[12] 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.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[15] Mozilla Foundation. Promise - JavaScript — mdn. `https://developer.mozilla.org/`

en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, Accessed Sep. 2022.

[16] Rust Foundation. Future in std::future - Rust. https://doc.rust-lang.org/std/future/trait.Future.html, Accessed Sep. 2022.

[17] The Linux Foundation. Zephyr project. https://www.zephyrproject.org/, Accessed Jan. 2024.

[18] The Futhark programming language high-performance purely functional data-parallel array programming on the GPU. https://futhark-lang.org/, 2020.

[19] Aditya Gupta and Amritpal Singh. A comprehensive survey on cyber-physical systems towards healthcare 4.0. *SN Computer Science*, Vol. 4, , feb. 2023.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1305–1320, 1991.

[21] Kevin Hammond, Christian Ferdinand, and Reinhold Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Rev.*, Vol. 3, No. 4, pp. 27–36, October 2006.

[22] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In *2nd International Conference on Generative Programming and Component Engineering (GPCE 2003)*, Vol. 2830 of *Lecture Notes in Computer Science*, pp. 37–56. Springer-Verlag, 2003.

[23] Kevin Hammond and Greg Michaelson. Predictable space behaviour in FSM-Hume. In *Implementation of Functional Languages (IFL 2002)*, Vol. 2670 of *Lecture Notes in Computer Science*, pp. 1–16. Springer, 2003.

[24] Caleb Helbling. Juniper language documentation (ver. 2.2.0). http://www.juniper-lang.org/language_docs.html, Nov. 2016.

[25] 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.

[26] Troels Henriksen. Towards Size Types in Futhark. https://futhark-lang.org/blog/2019-08-03-towards-size-types.html, August 2019.

[27] Troels Henriksen. Futhark 0.15.1 released - now with size types! https://futhark-lang.org/blog/2020-03-15-futhark-0.15.1-released.html, March 2020.

[28] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 556–571, New York, NY, USA, 2017. ACM.

[29] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. *SIGPLAN Not.*, Vol. 52, No. 1, pp. 359–373, January 2017.

[30] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, Vol. 38, No. 1, pp. 185–197, January 2003.

[31] 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.

[32] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, Vol. 37, No. 1–3, pp. 67–111, 2000.

[33] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space:

Towards embedded ml programming. *SIGPLAN Not.*, Vol. 34, No. 9, pp. 70–81, September 1999.

[34] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pp. 410–423. ACM, Jan. 1996.

[35] Facebook Inc. ReactJS: A JavaScript library for building user interfaces. `https://reactjs.org`, Accessed Sep. 2022.

[36] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pp. 354–369. Springer Berlin Heidelberg, 2009.

[37] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. A task-based dsl for microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. *SIGPLAN Not.*, Vol. 47, No. 1, pp. 45–58, January 2012.

[39] Edward A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369, 2008.

[40] Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. Tinytimber, reactive objects in c for real-time embedded systems. In *2008 Design, Automation and Test in Europe*, pp. 1382–1385, 2008.

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

[42] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, Vol. 193, pp. 29–45, 2007.

[43] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.*, Vol. 20, No. 4, may 2021.

[44] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. Interpreting task oriented programs on tiny computers. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Verimag lustre v6. `http://www-verimag.imag.fr/Lustre-V6.html`, 2020.

[46] 松村有倫, 渡部卓雄. 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. 情報処理学会論文誌 (プログラミング), Vol. 13, No. 2, pp. 1–13, apr 2020.

[47] 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. ACM, 2002.

[48] Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. Futures and promises — Scala documentation. `https://docs.scala-lang.org/overviews/core/futures.html`, Accessed Sep. 2022.

[49] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[50] 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.

[51] Raspberry Pi. Raspberry pi pico. `https://www.raspberrypi.com/products/raspberry-pi-pico/`, Accessed Jan. 2024.

[52] John H. Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, 1993.

[53] 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.

[54] 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.

[55] 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.

[56] Francisco Sant'Anna, Noemi Rodriguez, Roberto Ierusalimschy, Olaf Landsiedel, and Philippas Tsigas. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[57] Francisco Sant'Anna, Alexandre Sztajnberg, Ana Lúcia de Moura, and Noemi Rodrigues. Transparent standby for low-power, resource-constrained embedded systems: A programming language-based approach (short wip paper). In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2018, p. 94–98, New York, NY, USA, 2018. Association for Computing Machinery.

[58] 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. ACM, 2020.

[59] Abhiroop Sarkar, Bo Joel Svensson, and Mary Sheeran. Synchron - an api and runtime for embedded systems. 06 2022.

[60] 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.

[61] Amazon Web Services. FreeRTOS Real-time operating system for microcontrollers. `https://www.freertos.org/index.html`, Accessed Jan. 2024.

[62] Wang Sheng and Takuo Watanabe. Functional reactive EDSL with asynchronous execution for resource-constrained embedded systems. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Vol. 850 of *Studies in Computational Intelligence*, pp. 171–190. Springer, Aug. 2019.

[63] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pp. 1–6, 2011.

[64] Kazuhiro Shibanai and Takuo Watanabe. Distributed functional reactive programming on actor-based runtime. In *8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018)*, pp. 13–22. ACM, Nov 2018.

[65] SodiumFRP. SodiumFRP. `https://github.com/SodiumFRP`, Accessed Mar. 2023.

[66] Kento Sogo, Yuta Tsuji, Sosuke Moriguchi, and Takuo Watanabe. Periodic and aperiodic task description mechanisms in an FRP language for small-scale embedded systems. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2023, pp. 43–53, New York, NY, USA, 2023. ACM.

[67] Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. ReactiFi: Reactive programming of wi-fi firmware on mobile devices. *The Art, Science, and Engineering of Programming*, Vol. 5, No. 2, oct 2020.

[68] STMicroelectronics. Stm32 32-bit arm cortex mcus. `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html`, Accessed Jan. 2024.

[69] Espressif Systems. ESP32. `https://www.espressif.com/en/products/socs/esp32`, Accessed Jan. 2024.

[70] RxJS Team. RxJS: Reactive extensions library for JavaScript. `https://rxjs.dev`, Accessed Sep. 2022.

[71] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, Vol. 132, No. 2, pp. 109–176, February 1997.

[72] 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.

[73] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, pp. 86–101, Berlin, Heidelberg, 2003. Springer-Verlag.

[74] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional programming (ICFP 2001)*, pp. 146–156. ACM SIGPLAN, 2001.

[75] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Langauges*, Vol. 2257 of *Lecture Notes in Computer Science*, pp. 155–172. Springer-Verlag, 2002.

[76] 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.

[77] 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.

[78] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. Towards introducing asynchronous tasks to an FRP language for small-scale embedded systems. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2022, p. 1–12, New York, NY, USA, 2022. Association for Computing Ma-

chinery.

[79] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. Switching mechanism for update timing of time-varying values in an FRP language for small-scale embedded systems. In *Proceedings of the 2024 13th International Conference on Software and Computer Applications*, ICSCA '24, New York, NY, USA, 2024 (to appear). Association for Computing Machinery.

[80] Zélus a synchronous language with ODEs. `http://zelus.di.ens.fr/index.html`, 2020.

[81] Christoph Zenger. Indexed types. *Theoretical Computer Science*, Vol. 187, No. 1, pp. 147–165, 1997.

付録 A

$\mathrm{Emfrp}^{\mathrm{BCT}}$ に関する証明

A.1 使用メモリ量決定アルゴリズムの停止性の証明

定理 5 (Termination of the memory usage estimation algorithm)**.**
仮定

- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C}$
- (Γ, Δ)-consistent
- 式 e に現れる全ての関数 g は，$f \geq_{\mathcal{F}}^{*} g$ を満たす
- Δ は $\mathcal{C}$ のモデルである

のもとで，

- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!]$ は有限ステップで停止する
- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s, t, u)$ のとき，$\tau_{\mathcal{M}} = \tau$

が成り立つ．

証明. 三つ組 $(f, ev_{\Delta}[\![\overrightarrow{\delta}]\!], e)$ を定義する．f は関数名である．関数の順序を $\geq_{\mathcal{F}}^{*} \cup \{(-, f) \mid f \in \mathrm{dom}(\mathcal{F})\}$ によって定める．ただし $-$ はノードの定義式を検査する際に使用されるプレースホルダである．次に $\overrightarrow{\delta}$ は f の尺度関数である．最後に式 e について，e の部分式は e よりも小さいとして順序を定める．

停止性の証明はこの三つ組の辞書式順序についての帰納法で行う．型導出 $(\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C})$ の最後に適用された規則によって場合分けする．

最後に適用された型付け規則が T-CONST, T-VAR, T-NODE, T-ATLAST の場合は，言明は明らかに成り立つ．

T-CTOR, T-LET, T-IF, T-OP, T-ADJ, T-FIT の場合は，対象の式の部分式について，三つ組の最後の要素が減少しているので，帰納法の仮定より直ちに言明が導かれる．

T-CASE の場合，`case`式全体を e とする．ブランチの持つ式をそれぞれ e_i とする．e の分解対象の式については，三つ組の最後の要素が減少しているので，帰納法の仮定より言明が成り立つ．次に，それぞれのブランチについては，$\overline{\mathcal{M}}$ によって処理される．$\overline{\mathcal{M}}$ の内部では，コンストラクタに含まれるサイズ変数のためのサイズ割り当て集合 $\mathbf{A}$ を生成する．与えられたモデル Δ とこの割り当ての連接はブランチの式 e_i の型検査の際に抽出されるサイズ制約のモデルとなる．$\mathbf{A}$ は有限集合であるから，$\overline{\mathcal{M}}$ の内部ではブランチの持つ式 e_i について $\mathcal{M}$ が有限回呼び出される．e_i は e の部分式であるから，三つ組の最後の要素が減少するため，帰納法の仮定より言明が成り立つ．それゆえ，$\overline{\mathcal{M}}$ の実行も有限ステップで停止する．また，型付けによる e 全体の型と $\mathcal{M}(e)$ の結果として得られる型は同一である．したがって，e の検査対象式について言明が成り立ち，それぞれのブランチについても有限ステップで実

行が停止するため，e 全体としても言明は成り立つ．

T-CALL の場合，式の形は $g(e_1, \ldots, e_n)$ である．関数呼び出しの実引数式 $e_1, \ldots, e_n$ については，それぞれ $g(e_1, \ldots, e_n)$ の部分式であるから，三つ組の最後の要素が減少するため，帰納法の仮定より言明が成り立つ．また，結果の型は型付け規則と $\mathcal{M}$ とで同一のものが得られる．$g \neq f$ の場合，言明の仮定より $f \geq^*_{\mathcal{F}} g$ であるから，g 定義式を対象にした $\mathcal{M}$ の呼び出しでは，三つ組の 1 つ目の要素が減少している．よって帰納法の仮定より，言明が成り立つ．$g = f$ の場合，この関数呼び出しは再帰呼び出しである．三つ組の 2 番目の要素，つまり再帰関数呼び出しでは尺度が減少することを示す．言明の仮定から Δ は $\bigwedge_{i \in 1 \ldots n} \mathcal{C}_i \wedge \bigwedge_{i \in 1 \ldots m} (\theta \mathcal{A}_i) \wedge \mathcal{R}$ のモデルなので，$\mathcal{R}$ のモデルでもある．したがって $\mathcal{R}$ の制約より，$ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}}(\theta\delta)]\!] < ev_\Delta[\![\overrightarrow{\delta}]\!]$ である．θ と $\overrightarrow{\delta}$ の定義から，ある i について，$\theta\delta = \tau_i \downarrow$ が成り立つ．g の定義式を走査する $\mathcal{M}$ での尺度の値は，$ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\overrightarrow{\delta}]\!]$ である．

$$
\begin{aligned}
& ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\overrightarrow{\delta}]\!] \\
&= ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}} \pi_i \downarrow]\!] \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}}(ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\pi_i \downarrow]\!]) \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}}(ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![ev_\Delta[\![\tau_i \downarrow]\!]]\!]) \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}}(ev_\Delta[\![\tau_i \downarrow]\!]) \\
&= \Sigma_{\delta \in \overrightarrow{\delta}}(ev_\Delta[\![\theta\delta]\!]) \\
&= ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}}(\theta\delta)]\!] \\
&< ev_\Delta[\![\overrightarrow{\delta}]\!]
\end{aligned}
$$

以上の式変形により，関数の定義式のために呼び出される $\mathcal{M}$ での尺度は，もとの式の尺度と比べて減少している．したがって，三つ組の 2 番目の要素が減少するので，帰納法の仮定より言明が成り立つことがわかった．関数名についての場合分けのどちらの場合にも，関数定義式の $\mathcal{M}$ の実行は停止する．したがって，言明は成り立つ． □

A.2 式の型付けに対する健全性

定理 7 (Soundness of typing of expression)**.**
仮定

- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid \mathcal{C}$.
- $(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent.
- 式 e に現れる全ての関数名 g は，$f \geq^*_{\mathcal{F}} g$ を満たす
- Δ は $\mathcal{C}$ のモデルである
- $\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!] = (\tau_{\mathcal{M}}, s_{\mathcal{M}}, t_{\mathcal{M}}, u_{\mathcal{M}})$.

のもとで，すべての $s \geq s_{\mathcal{M}}, t \geq t_{\mathcal{M}}, u \geq u_{\mathcal{M}}$ について，ある l, t', H', σ が存在し，

- $[s]E \mid [t]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l; [t']H'$
- $t' \geq t - t_{\mathcal{M}}$
- $H', \mathcal{T} \vdash l : \sigma$
- $\sigma \downarrow \leq ev_\Delta[\![\tau \downarrow]\!]$
- σ と τ のそれぞれのサイズパラメータを除いた型名が一致する

を満たす．

証明．証明は，定理 5 の証明で用いた三つ組の辞書式順序についての帰納法で示す．型導出の最後に適

用した規則によって場合分けする．

最後に適用した規則が，T-CONST の場合，$H' = H, l \mapsto c^{\mathcal{B}}$，$t' = t - 1$，$\sigma = \mathcal{B}$ とすればよい．

T-VAR，T-NODE，T-ATLAST の場合，$H' = H$，$t' = t$，σ はそれぞれの型とすればよい．

T-CTOR の場合，$H = H_0$，$t = t_0 + 1$，$e = \chi(e_1, \ldots, e_n)$ とする．$1 \leq i \leq n$ について，$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!] = (\tau_{\mathcal{M}i}, s_{\mathcal{M}i}, t_{\mathcal{M}i}, u_{\mathcal{M}i})$ とする．$1 \leq i \leq n$ について，$\mathcal{M}$ の定義から，$s \geq s_{\mathcal{M}} = 1 + \bigsqcup_{j \in 1\ldots n} s_{\mathcal{M}j} > s_{\mathcal{M}i}$，つまり，$s - 1 \geq s_{\mathcal{M}i}$ が成り立つ．同様に，$u \geq u_{\mathcal{M}} \geq u_{\mathcal{M}i}$ が成り立つ．さらに，$t = t_0 + 1 \geq t_{\mathcal{M}} = 1 + \Sigma_{i \in 1\ldots n} t_{\mathcal{M}i} \geq t_{\mathcal{M}1}$ だから，$t_0 \geq \Sigma_{i \in 1\ldots n} t_{\mathcal{M}i} \geq t_{\mathcal{M}1}$ が成り立つ．これらの条件により帰納法の仮定から，$[s-1]E \mid [t_0]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1$ かつ $t_1 \geq t_0 - t_{\mathcal{M}1} \geq \Sigma_{i \in 2\ldots n} t_{\mathcal{M}i}$ かつ $H_1, \mathcal{T} \vdash l_1 : \sigma_1$ かつ $\sigma_1 \downarrow = ev_{\Delta}[\![\tau_1 \downarrow]\!]$ を満たし，t_1 と σ_1 のサイズパラメータを除いた型名が一致するような t_1, l_1, H_1, σ_1 が存在する．これをコンストラクタの引数の式について繰り返すと，$t_n \geq t_0 - \Sigma_{i \in 1\ldots n} t_{\mathcal{M}i} \geq 0$ と H_n を得る．ここで，操作的意味論の規則 E-CTOR を適用すると結果として，$t' = t_n \geq t_0 - \Sigma_{i \in 1\ldots n} t_{\mathcal{M}i} = (t_0 + 1) - (1 + \Sigma_{i \in 1\ldots n} t_{\mathcal{M}i}) = t - t_{\mathcal{M}}$，$H' = H_n, l \mapsto \chi[k](l_1, \ldots, l_n)$ を得る．

T-LET の場合，式の形は `let` x `=` e_1 `in` $e2$ である．$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!] = (\tau_{\mathcal{M}1}, s_{\mathcal{M}1}, t_{\mathcal{M}1}, u_{\mathcal{M}1})$ とする．$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma,x:\tau_1}[\![e_2]\!] = (\tau_{\mathcal{M}2}, s_{\mathcal{M}2}, t_{\mathcal{M}2}, u_{\mathcal{M}2})$ とする．$s_{\mathcal{M}} \geq s_{\mathcal{M}1}, u_{\mathcal{M}} \geq u_{\mathcal{M}1}$ かつ $s_{\mathcal{M}} \geq 1 + s_{\mathcal{M}2}, u_{\mathcal{M}} \geq u_{\mathcal{M}2}$ である．$t_{\mathcal{M}} = t_{\mathcal{M}1} + t_{\mathcal{M}2}$ である．この時，$s \geq s_{\mathcal{M}}, t \geq t_{\mathcal{M}}, u \geq u_{\mathcal{M}}$ とする．e_1 について帰納法の仮定から，$[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1$ かつ $t_1 \geq t - t_{\mathcal{M}1}$ かつ $H_1, \mathcal{T} \vdash l_1 : \sigma_1$ かつ $\sigma_1 = ev_{\Delta}[\![\tau_1 \downarrow]\!]$ を満たす l_1, t_1, H_1, σ_1 を得る．また，$H \subseteq H_1$ であるから，$(\mathcal{L}, (E, x \mapsto l_1), H_1, (\Gamma, x : \tau_1), \Delta)$-consistent が成り立つ．よって e_2 について帰納法の仮定より，$[s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2$ かつ $t_2 \geq t_1 - t_{\mathcal{M}2}$ かつ $H_2, \mathcal{T} \vdash l_2 : \sigma_2$ かつ $\sigma_2 = ev_{\Delta}[\![\tau_2 \downarrow]\!]$ を満たす l_2, t_2, H_2, σ_2 を得る．したがって，$t' = t_2 \geq t - (t_{\mathcal{M}1} + t_{\mathcal{M}2})$，$l' = l_2$，$H' = H_2$，$\sigma = \sigma_2$ を得る．

T-IF，T-OP，T-ADJ の場合は T-CTOR と同様である．T-FIT の場合は T-LET と同様である．T-CALL の場合は，実引数の式については T-CTOR と同様に扱う．次に，引数のサイズ変数に関する一貫性 (consistent) については T-LET と同様に扱う．最後に，関数の定義式については定理 5 での関数呼び出しと同様の手法で．三つ組の減少を示すことができる．これらを組み合わせて関数呼び出し式全体の言明が示される．

T-CASE の場合，分解対象の式を e_0 とする．$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_0]\!] = (\tau_{\mathcal{M}0}, s_{\mathcal{M}0}, t_{\mathcal{M}0}, u_{\mathcal{M}0})$ とする．ブランチを $1 \leq i \leq n$ について $branch_i$ とし，$\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(branch_i) = (s_{\overline{\mathcal{M}i}}, t_{\overline{\mathcal{M}i}}, u_{\overline{\mathcal{M}i}})$ とする．この時，$s_{\mathcal{M}} \geq s_{\mathcal{M}0}$ かつ $1 \leq i \leq n$ について $s_{\mathcal{M}} \geq s_{\overline{\mathcal{M}i}}$ である．また，$u_{\mathcal{M}} \geq u_{\mathcal{M}0}$ かつ $1 \leq i \leq n$ について $u_{\mathcal{M}} \geq u_{\overline{\mathcal{M}i}}$ である．さらに，$t_{\mathcal{M}} = t_{\mathcal{M}0} + \Sigma_{i \in 1\ldots n} t_{\overline{\mathcal{M}i}}$ である．$s \geq s_{\mathcal{M}}, t \geq t_{\mathcal{M}}, u \geq u_{\mathcal{M}}$ とする．ここで e_0 について帰納法の仮定から，$[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_0 \Downarrow_u l_0; [t_0]H_0$ かつ $t_0 \geq t - t_{\mathcal{M}0}$ かつ $H_0, \mathcal{T} \vdash l_0 : \sigma_0$ かつ $\sigma_0 \downarrow = ev_{\Delta}[\![\rho^{\psi} \downarrow]\!]$ を満たす t_0, l_0, H_0, σ_0 が存在し，$H_0(l_0) = \chi[k](l_1, \ldots, l_m)$ である．$\mathcal{M}$ の内部ではブランチは $\overline{\mathcal{M}}$ によって探索が行われる．$\overline{\mathcal{M}}$ では，コンストラクタに現れる再帰しているサイズ変数への全ての割り当て集合 $\mathbf{A}$ を生成する．また，固定サイズのサイズ変数に対しても割り当て b を生成する．コンストラクタで導入されるサイズ変数に対するサイズ制約を $\mathcal{C}$，ブランチの式を型付けした際に得られるサイズ制約 $\mathcal{C}'$ とすると，$\{(\Delta, a, b) \mid a \in A\}$ は $\mathcal{C} \wedge \mathcal{C}'$ を満たす全てのモデルの集合である．したがって，$d \in \{(\Delta, a, b) \mid a \in A\}$ について，$(\mathcal{L}, (E, \{x_i \mapsto l_i\}_{i \in 1\ldots m}), H_0, (\Gamma, \{x_i : \pi_i\}_{i \in 1\ldots m}), d)$-consistent が成り立つ．それゆえ，ブランチの式について帰納法の仮定を用いることができ，T-LET の場合と同様の手順で言明を示すことができる．

□

付録 B

EvEmfrp/S の生成コード

B.1 BlinkLEDHybrid モジュール

```
#define swap(a, b) do{void* p=a;a=b;b=p;}while(0)
#define T_INF UINT32_MAX
#define WALL_CLOCK_MAX (T_INF/2)

union interrupt_events_t {
  uint32_t bits; struct { unsigned Eedge : 1; };
};

union timer_events_t {
  uint32_t bits;
  struct { unsigned Eedge_en : 1;
           unsigned Eend : 1;
           unsigned Psample : 1;  };
};

union when_events_t {
  uint32_t bits; struct { unsigned Eout : 1; };
};

// Event occurrence flags
struct event_list_t {
  union interrupt_events_t i;
  union timer_events_t t;
  union when_events_t w;
} el;

// Record events in interrupt functions
volatile union interrupt_events_t events_buf;

// Wall clock (10kHz)
volatile uint32_t wall_clock;

// Time when the next timer event occurs
uint32_t occur_time[3]; // 0:Eedge_en,1:Eend,2:Psample
uint32_t next_occur_time;

// Current period of periodic timing
uint32_t period[1]; // 0:Psample

// Nodes
bool *btn_status, btn_status_mem[1];
bool *btn, *btn_last, btn_mem[2];
bool *btn_on, *btn_on_last, btn_on_mem[2];
bool *push_pulse, push_pulse_mem[1];
bool *led, *led_last, led_mem[2];
```

```
46
47 void init_node(void){
48   btn_status = &btn_status_mem[0];
49   btn = &btn_mem[0];
50   btn_last = &btn_mem[1];
51   btn_on = &btn_on_mem[0];
52   btn_on_last = &btn_on_mem[1];
53   push_pulse = &push_pulse_mem[0];
54   led = &led_mem[0];
55   led_last = &led_mem[1];
56
57   *btn_last = false;
58   *btn_on_last = false;
59   *led_last = false;
60 }
61
62 // Micro iterations
63 void miter_Eedge(uint32_t current_time){
64   // Set the event to enable interrupts(Eedge_en)
65   occur_time[0] = current_time + 10000; // 1000ms
66   // Update nodes
67   *btn = false;
68   swap(btn, btn_last);
69   // Set up another timing that is affected by Eedge(Eend, Psample)
70   occur_time[1] = current_time + 50000; // 5000ms
71   el.t.Psample = 1; // phase = 0
72   occur_time[2] = current_time + 0;  // phase = 0
73   period[0] = 200; // 20ms
74 }
75
76 void miter_Eend(uint32_t current_time){
77   // Set the event to disable
78   occur_time[1] = T_INF;
79   // No Node Updating
80   // Set up another timing that is affected by Eend(Psample)
81   el.t.Psample = 0; // phase = INF
82   occur_time[2] = T_INF; // phase = INF
83   period[0] = T_INF; // period = INF
84 }
85
86 void miter_Psample(uint32_t current_time){
87   // Set the event for the next occurrence of Psample
88   occur_time[2] = current_time + period[0];
89   // Update nodes
90   Input_btn_status(btn_status);
91   *btn = *btn_status;
92   *btn_on = *btn_last && *btn;
93   *push_pulse = !(*btn_on_last) && *btn_on;
94   swap(btn, btn_last);
95   swap(btn_on, btn_on_last);
96   // Set up another timing that is affected by Psample(Eout)
97   if(*push_pulse) el.w.Eout = 1;
98 }
99
100 void miter_Eout(uint32_t current_time){
101   // Update nodes
102   *led = !(*led_last);
103   Output_led(led);
104   swap(led, led_last);
105 }
106
107 // Activation for main module
108 void ActivateBlinkLEDHybrid(void){
```

```
109  // Initialize
110  INIT_TIMER(10000); // 10kHz timer
111  INIT_INTERRUPT();
112  init_node();
113
114  // Set initial phase for timer events
115  occur_time[0] = 0; // enable Eedge
116  occur_time[1] = T_INF; // Eend
117  occur_time[2] = T_INF; // Psample
118
119  // Set initial period for periodic events
120  period[0] = T_INF;
121
122  // reset wall clock
123  wall_clock = 0;
124
125  while(1){ // Iteration
126    while(1){ // Micro Iterations
127      el.w.bits = el.t.bits = 0;
128      next_occur_time = occur_time[0];
129
130      // Finding events to handle
131      for(int i=0;i<3;i++){
132        next_occur_time =
133          min(next_occur_time, occur_time[i]);
134        if(occur_time[i] <= wall_clock)
135          el.t.bits |= (1 << i);
136      }
137
138      // Wall clock overflow prevention
139      if(wall_clock >= WALL_CLOCK_MAX){
140        uint32_t offset =
141          min(next_occur_time, wall_clock);
142        wall_clock -= offset;
143        if(next_occur_time != T_INF)
144          next_occur_time -= offset;
145        for(int i=0;i<3;i++)
146          if(occur_time[i] != T_INF)
147            occur_time[i] -= offset;
148      }
149
150      // Disable all interrupts and get interrupt events
151      DISABLE_INTERRUPT();
152      el.i.bits = events_buf.bits;
153      events_buf.bits = 0;
154
155      if(el.i.bits | el.t.bits) ENABLE_INTERRUPT();
156      else break; // Go to sleep if no events to handle
157
158      // Start micro iteration time measurement
159      START_ONESHOT_TIMER();
160
161      // Interrupt enable event
162      if(el.t.Eedge_en) {
163        occur_time[0] = T_INF;
164        ENABLE_ONESHOT_INTERRUPT_Eedge();
165      }
166
167      // Micro iterations for each event
168      if(el.i.Eedge)
169        miter_Eedge(wall_clock + GET_TIMER_COUNT());
170      if(el.t.Eend)
171        miter_Eend(wall_clock + GET_TIMER_COUNT());
```

```
172       if(el.t.Psample)
173         miter_Psample(wall_clock + GET_TIMER_COUNT());
174       if(el.w.Eout)
175         miter_Eout(wall_clock + GET_TIMER_COUNT());
176
177       // Update wall clock
178       wall_clock += GET_TIMER_COUNT();
179       // End micro iteration time measurement
180       STOP_ONESHOT_TIMER();
181     }
182
183     uint32_t sleep_time = next_occur_time - wall_clock;
184     // Check if there is no timer event.
185     // If not, it can sleep forever.
186     bool forever = (next_occur_time == T_INF);
187     ENABLE_INTERRUPT();
188     // Obtain the actual sleep time,
189     // since it may be woken up by an interrupt event.
190     uint32_t actual_sleep_time =
191         SLEEP_AND_AWAKE(forever, sleep_time);
192     // No need to advance wall clock without timer event
193     if(!forever) wall_clock += actual_sleep_time;
194   }
195 }
196
197 // Call in interrupt function for Eedge event
198 void OCCURRED_Eedge(void) {
199   DISABLE_ONESHOT_INTERRUPT_Eedge();
200   events_buf.Eedge = 1; }
201
202 // Skelton codes for platforms
203 void INIT_INTERRUPT(void){ ... }
204 void ENABLE_INTERRUPT(void){ ... }
205 void DISABLE_INTERRUPT(void){ ... }
206 void INIT_TIMER(uint32_t frequency){ ... }
207 void START_ONESHOT_TIMER(void){ ... }
208 void STOP_ONESHOT_TIMER(void){ ... }
209 uint32_t GET_TIMER_COUNT(void){ ... }
210 uint32_t SLEEP_AND_AWAKE(bool forever,uint32_t sleep_time){ ... }
211
212 // Skelton codes for the application
213 void ENABLE_ONESHOT_INTERRUPT_Eedge(void){...}
214 void DISABLE_ONESHOT_INTERRUPT_Eedge(void){...}
215 void IRQ_HANDLER_Eedge(void){ OCCURRED_Eedge(); }
216 void Input_btn_status(bool *btn_status){...}
217 void Output_led(bool* led){...}
218
219 // main function
220 int main(void){
221   init_sensors_actuators();
222   init_timer();
223   init_interrupt();
224   ActivateBlinkLEDHybrid();
225   return 0;
226 }
```

ソースコード B.1　BlinkLEDHybridモジュールから生成される C 言語コード (図 5.8)