日本ソフトウェア科学会第22回プログラミングおよび
プログラミング言語ワークショップ(PPL 2020), 2020年3月

小規模組込みシステム向けFRP言語に対する再帰的データ型の導入

横山 陽彦 [1]　森口 草介 [2]　渡部 卓雄 [3]

東京工業大学 情報理工学院

[1] akihiko@psg.c.titech.ac.jp

[2] chiguri@acm.org

[3] takuo@acm.org

概 要　関数リアクティブプログラミング (FRP) 言語 Emfrp に対する再帰的なデータ型の導入手法を提案する．Emfrp はマイクロコントローラのような小規模システム向けに設計された純粋 FRP 言語である．この言語では，関数やデータ型の再帰的な定義を禁止する等の言語的制約を課すことで，時変値の更新処理が停止することや実行時に必要なメモリサイズを静的に決定できることが保証される．本研究では型に対してサイズの情報を含めることで，Emfrp の性質を維持しつつこの制約を緩和する手法を提案する．

1　はじめに

関数リアクティブプログラミング (Functional Reactive Programming, FRP) は，時間とともに連続的あるいは離散的に変化する値を抽象化した**時変値 (time-varying value)** を用いることで，組込みシステムや GUI に代表されるリアクティブシステムの効果的な記述を支援するプログラミングパラダイムである．時変値にもとづく FRP の考え方は，Elliott らによる Haskell 用の対話的アニメーションライブラリ Fran[4] において最初に導入され，その後 Yampa[9] や Reactive Banana[1] などの Haskell 用 FRP ライブラリや，主に Web アプリケーションのための言語 Elm[3][1] など，様々な FRP 言語，DSL，ライブラリで用いられている．

組込みシステムのプログラムでは，リアクティブな動作を実現するためにポーリングや割り込み（およびそれらの混在）が多用され，プログラムの見通しを悪くしている．我々は，小規模組込みシステムを対象とする FRP 言語 Emfrp[13] を設計・実装し，いくつかの例題を通してその組込みシステム開発における有用性を明らかにしてきた．

Emfrp は副作用を持たない静的型付き FRP 言語である．マイクロコントローラのような制限されたリソースのもとで実行されるプログラムの記述を目的として設計されており，実際に RAM が 2.5KB 程度の 8 ビットマイクロコントローラ上で動作するロボットの制御プログラムなどの記述が可能である．そのため，実行時のメモリ使用量を抑えるとともに，特に実行時のメモリ使用量を静的に決定できるよう，時変値を第一級オブジェクトとしないことや，関数やデータ型における再帰的定義の禁止など，言語設計上の制約がいくつか導入されている．これらの制約のもとでも多くの応用プログラムの記述は可能であるが，文字列やリスト，木など，小規模な組込みシステムにおいても有用なデータ構造の自然な表現が困難になるという問題がある．

本研究では，構造のサイズに関するパラメータを型として付与することで，関数やデータ型の再帰的定義の禁止を緩和する．提案手法では，通常の型検査に加えオブジェクトのサイズに関する制約を検査することで，オブジェクトが制限なく増え続けるようなプログラムやノード更新計算の停止性が保証されないプログラムの検出を行う．新たに導入した再帰データ型は「そのオブジェクト

[1] バージョン 0.17 以降は FRP 言語ではなくなっている [2].

```
1  module Distance                ## モジュール名
2  in  vl : Float, vr : Float,    ## 左右の車輪の対地速度 (m/sec)
3      theta : Float,             ## ロボットの向きとx 軸のなす角度 (rad)
4      t(0)  : Int                ## 起動時からの経過時間 (msec)
5  out x     : Float              ## x 軸方向の原点からの位置 (m)
6  use Std                        ## 使用ライブラリ
7
8  ## 直前値との差によって微小変化を表現
9  node dt = (t - t@last) / 1000.0   ## 微小時間 (sec)
10
11 ## 直前値との和によって累積値 ( 積分 ) を表現
12 node init[0.0] x = x@last + (vr + vl) * cos(theta) * dt / 2
```

プログラム 1. 例題 : 2 輪ロボットの位置

内に自身と同じ構造が最大で何個出現するか」をパラメータとして持ち，実行時に必要なデータサイズの静的な決定を可能にする．また関数については，呼び出し時に引数に渡される値の構造的な単調減少を検査することで，計算の停止性が保証できる場合に再帰関数の定義を許可する．これらの拡張により，Emfrp の持つ実行時のメモリ使用量を静的に決定できるという特徴を保ちつつ，再帰的なデータ構造の自然な定義や使用を可能にした．本稿では左偏ヒープ木を扱うプログラムを例として取り上げ，サイズに関するパラメータや再帰関数の定義についての説明を行う．

以下，第 2 節で Emfrp の概要と，本研究のモチベーションとなった構文的制約に伴う問題点について述べる．続いて，第 3 節で提案手法となる型システムについて例と共に説明し，第 4 節でその FRP 言語への導入について述べる．そして既存研究との関連について第 5 節で延べ，第 6 節でまとめと今後の課題について説明する．

2 関数リアクティブプログラミング言語 Emfrp

Emfrp[13] は小規模組込みシステム向けに設計された関数リアクティブプログラミング言語である．本節では Emfrp の概要を例を通して説明し，本研究の動機となる問題点について述べる．言語の詳細については文献 [13] およびソースコードリポジトリ[2]に付随する Wiki を参照されたい．

2.1 言語の概要

プログラム 1 は 2 輪ロボットの位置を計算するプログラムであり，文献 [9] において Yampa（Haskell 用の FRP ライブラリ）の例題として挙げられているものを Emfrp で書き直したものである．以下，この例を用いて Emfrp の概要を説明する．

このプログラムは左右の車輪それぞれの対地速度 v_l と v_r，ロボットの向きと x 軸がなす角度 θ，および時刻 t を入力とし，現在の x 軸方向のロボットの位置 x を出力とする．これら v_l, v_r, θ, x は時刻 t の関数であり，以下の式 (1) で表される関係を持つ [9]．

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

Emfrp のプログラムはモジュールと呼ばれる単位で構成される．プログラム 1 は `Distance` という名前を持つ 1 個のモジュールからなる．モジュールのヘッダ部（1–6 行目）ではモジュール名や入出力ノードおよび使用するライブラリが宣言され，本体（8 行目以降）では型，定数，関数およびノード（後述）が定義される．

この例題では v_l, v_r, θ, x および t を時変値として表している．Emfrp では時変値はノードと呼ばれる．ノードは連続値あるいは離散値をとることができる．プログラム 1 では `vl`, `vr`, `theta`, `t`, `x` お

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

よび dt がノードであり，それぞれ v_l, v_r, θ, t, x および Δt（時間の変化量）を表している．ノードは外部機器に接続されることを想定している入力・出力ノード，およびそのような接続を想定しない内部ノードの 3 種類に分類される．例題では vl, vr, theta および t が入力ノード，x が出力ノード，そして dt が内部ノードである．

入力ノードの値は接続されている外部機器によって決定される．プログラム 1 では，vl および vr の値が左右それぞれの車輪に接続された回転計（ロータリーエンコーダ）の計測値から得られる対地速度を，theta の値が方位センサの計測値から得られる x 軸との角度を表している[3]．また t はシステムのタイマと接続され，起動時からの経過時間（ミリ秒単位）を表している．

入力ノード以外，つまり出力ノードおよび内部ノードの値はモジュールの本体において **node** n = e（あるいは **node init**[c] n = e）という構文によって定義される．ここで n はノード名，e は式であり，**init**[c] はノードの初期値（後述）が定数 c であることを表す．プログラム 1 では 8 行目および 10 行目がそれぞれ dt および x の定義である．各ノードの値は = の右辺の式（ノードの定義式と呼ぶ）によって決まる．すべての式は静的に型付けされるが，型推論機構をもつためノード定義において型を明示する必要はない．

ノード名の後に演算子@last が付いたものはそのノードの**直前値**，すなわち現在の値に至る直前（微小時間前）の値を表している．この例では dt および x の定義において用いられている．この例が示すように，演算子@last を用いることで変化量や累積値を容易に表現することができ，式 (1) のような時間積分で表される値の近似値を簡潔に表すことができる．他にも，各種の状態依存動作の記述に用いることが可能である [11, 13, 16]．

直前値の参照が行われるそれぞれのノードについては初期値の指定が必要である．これはプログラムの実行開始時における直前値として用いられる．例えばプログラム 1 では，ノード xの定義において **init**[0.0]で初期値が指定され（10 行目），tのような入力ノードについては，ヘッダ部において t(0)のようにして初期値が指定される（4 行目）．

2.2 実行モデル

Emfrp の実行モデルについて簡単に説明する．あるモジュールの本体において定義されたノード n について，その定義式 e 中にノード n' が演算子@lastを伴わずに出現するとき，n は n' に依存するという．この依存関係を n' から n への有向辺として有向グラフを構成できるが，Emfrp ではこのグラフは有向非巡回グラフ (DAG) であり，かつ入力ノードの入次数は 0 であることが要求される．第 2.3.1 節で後述するように，Emfrp ではノードは一級データではなく必ず定義時の名前で参照され，またノードを値とするようなノード（高階時変値）も存在しない．よってノード間の依存関係を表す DAG は静的に定まり，実行時に変化することはない．

Emfrp コンパイラはこの DAG についてトポロジカルソートを行い，得られたノードの列に沿って値の更新を行うコードを生成する．プログラム 1 の場合，例えば (t, dt, vl, vr, theta, x) のような列を得ることができる．この列に沿った 1 回の更新（この例の場合 tから xまで）を**更新サイクル**と呼ぶ．Emfrp の実行系は更新サイクルを繰り返すことでリアクティブな動作を実現する．また，ノード n の直前値 n@lastは前回の更新サイクルで得られた n の値として実現されている．

2.3 構文的制約とその問題点

第 1 節で述べたように Emfrp はマイクロコントローラのような小規模なシステム上で動作するプログラムを記述するために設計されている．そのために，(1) ノードの非一級データとしての表現，および (2) 関数定義およびデータ構造における再帰の禁止という構文的制約により実行時に使用するメモリ量をコンパイル時に決定できるようにしている．以下でこれらについて説明する．

[3] これらセンサの計測値をそのまま入力ノードの値とし，Emfrp のコード内で必要な補正を行うことも可能である．ここでは文献 [9] と同様に，例題を簡潔にするために補正後の値を用いている．

2.3.1 ノードの非一級データとしての表現

Emfrp ではノード（時変値）を一級データとして扱わない．つまり，ノードを変数に代入したり，ノードが関数の引数や戻り値となることはない．ノードは予約語 `node` を用いて定義されたもの（入力ノードについてはヘッダ部で定義されたもの）のみが用いられ，実行時にノードが増減することはない．また，ノードを値とするノード（高階時変値）の利用や，ノードの定義式が実行時に変化することはない．加えて，ノードは常に定義時の名前によってのみ参照される．関数の引数としてノード名が与えられた場合，関数に渡されるのは時変値ではなく呼び出しの時点におけるそのノードの値となる．

以上の制約により，ノードを表すデータ構造をヒープ領域ではなくスタティック領域に置くことが可能になり，ノードの値を更新するプログラムも簡潔になる．そのため，リソースに制約のある環境で動作する C プログラムへの翻訳が容易になる．また，ノードに対する操作は現在値と直前値の参照のみであり，時間漏れ (time-leak) や空間漏れ (space-leak) といった現象は発生しない．さらに，ノードの参照が宣言時の名前のみによって行われるため，ノード名を含む式において，(ノード以外の) 値の時変値への変換を暗黙的に行うことが容易である．これは，例えばアロー化 (Arrowized)FRP システム [9] において通常の関数のシグナル関数への変換やシグナル関数のアロー演算子による合成を明示的に行う（あるいはアロー構文を使う）必要があることと対照的であり，プログラムの作成や理解を容易にする．

前述したように Emfrp ではノード間の依存関係が実行時に変化することはないが，その一方で実行時文脈や状態に依存する動作の記述が（条件式を多用する等）煩雑になる．現在までに我々は文脈指向プログラミング機構 [16] や状態依存動作のモジュール化機構 [11] を提案し，この問題を部分的に解決している．

2.3.2 関数定義およびデータ構造における再帰の禁止

Emfrp ではモジュールの本体において関数や代数的データ型の定義が可能であるが，これらにおいて再帰的な定義を禁止している．その理由は第 2.2 節で述べた 1 回の更新サイクルが有限時間で停止することを保証したいためである．Emfrp は純粋な関数プログラミング言語であり，(while 文に相当する) 繰り返しを用いることはできない．したがって，再帰的な関数を禁止することで式の評価が有限時間で停止することが保証され，結果として 1 回の更新サイクルも有限時間で停止することになる[4]．

再帰的なデータ型の定義を禁止しているのは実行時に使用するメモリ量をコンパイル時に決定できるようにするためである．Emfrp では整数や浮動小数点数などの基本型に加え，組 (tuple) と代数的データ型を用いることができる．代数的データ型において再帰的な定義を禁止することで，データの大きさはコンパイル時に決定できることになる．

ただしこの制約のため，文字列やリスト，木など，小規模な組込みシステムにおいても有用なデータ構造の自然な表現が困難になる．そこで本研究では，実行時に使用するメモリ量をコンパイル時に確定できるという条件のもとで，再帰的なデータ型を導入する手法を提案する．

3 提案手法

本研究で導入する再帰的なデータ型は，データの種類をあらわす型の他に「そのオブジェクト内に自身と同じ構造が最大何個出現するか」というパラメータを保持する．このパラメータをサイズパラメータと呼ぶ．サイズパラメータは 0 より大きい正の数である．例えば，型 `IntList #5` は長さ

[4] Emfrp では入出力や組込み関数において C で記述されたコードを呼び出すことができるが，それらの停止性についてはここでは考慮の対象としていない．

```
## 左偏ヒープ (再帰データ型定義)
rectype Heap = E(Unit)                  ## 末端
             | T(Int, Int, Heap, Heap) ## ノード (ランク, 値, 左, 右)
```

プログラム 2. 再帰データ型の定義

が最大で 5 までのリスト構造であることを意味する．型 `IntTree #10` と書く場合には要素を最大で 10 個保持できるような木構造であることを表す．ここで，サイズパラメータはコンストラクタの最大ネスト回数を表しているわけではないということに注意されたい．再帰的なデータ型が引数や結果の型に現れる関数定義は，それぞれのサイズパラメータに関する制約を関数定義とともに記述する必要がある．また，再帰的なデータ型の導入に際し，Emfrp では禁止されていた再帰関数の定義を制限付きで導入する．

Emfrp に対して再帰的なデータ型を導入する上で維持すべき特徴は次の二つである．一つ目は実行時に必要なメモリ量を静的に決定できること，二つ目は各ノードの更新処理の停止性が保証されていることである．以降，型にサイズパラメータを付与することで，これらの特徴を保ちつつ再帰的なデータオブジェクトが導入できることを示す．

3.1 例題：左偏ヒープの実装

以下，提案手法で導入する再帰的なデータ型の定義と使用について，例題を通して概要を説明する．例題の完全なコード（モジュール定義）を付録 A に示す．

3.1.1 問題設定

「いままでに入力されたデータのうち大きさ上位 10 番目までの和」を出力するモジュールを例に取り上げる．センサなどから取得した値を入力ノード `x` で，内部データの合計値を出力ノード `y` で表す．どちらも値の型は整数（`Int`）とする．上位 10 番目までの数値を保持するため，モジュールは内部で左偏 (leftist) ヒープ [12] を保持する．入力されたデータがヒープ中の最小値よりも大きい場合に最小値を削除し，入力データを挿入することで常に上位 10 番目までの数値を保持することができる．

3.1.2 再帰的データ型の定義

左偏ヒープは再帰的データ型を用いて定義される．再帰的なデータ型の定義はプログラム 2 のように行う．

`rectype` キーワードを用いて再帰的なデータ型であることを明示する．型定義の際にはサイズパラメータは出現しないことに注意されたい．ただし，コンストラクタは内部的に関数として扱われるが，その際に `E` は型 `forall #n.(Unit)->Heap #n`，`T` は型 `forall #n,#m,#r.{#r=1+#n+#m}=>(Int,Int,Heap #n,Heap #m)->Heap #r` を持つように変換される．`T` にはサイズパラメータについての制約 $r = 1+n+m$ がかけられている．型式や制約式などの詳細な型システムについては後述の第 4.2 節で説明する．

3.1.3 関数定義

関数定義の際には通常の型検査に加え，引数と結果の型から得た制約と関数の式から得られる制約が帰結の関係であることを検査する．例として左偏ヒープを操作するための関数を定義する．木のランクを取得する関数はプログラム 3 のように定義され，型 `forall #n.(Heap #n)->Int` をもつ．この時サイズパラメータ`#n` には $n > 0$ 以外の制約はない．

この関数定義が適切であることを確認するために，まずサイズパラメータを無視した型検査を行う．その後，関数定義の右辺式が満たすべき制約を計算する．`case` 式の `E` のマッチについては制約はなく，その後の式もサイズパラメータについての制約はないため得られる制約は $\top$ である．`T` の

```
11 ## 木のランクを取得する: forall #n. (Heap #n) -> Int
12 fun rank(e: Heap #n) : Int = case e of E -> 0 | T (r, x, a, b) -> r
```

プログラム 3. 木のランクを取得する関数（通常の関数定義の例）

```
14 ## ヒープを構築する（値を 2つの木の上に追加する）
15 ## forall #n,#m. (Int, Heap #n, Heap #m) -> Heap #r
16 fun make(x: Int, a: Heap #n, b: Heap #m) : Heap #r where (#r = 1 + #n + #m) =
17     let ra = rank(a) in
18     let rb = rank(b) in
19     if ra > rb then T #r (rb+1, x, a, b) else T #r (ra+1, x, b, a)
```

プログラム 4. ヒープを構築する関数（サイズパラメータに関する制約を伴った関数定義の例）

マッチにより分解された a と b がそれぞれ Heap #p 型，Heap #q 型だとすると，コンストラクタ T の制約から $\forall p, q.n = 1 + p + q \to C$ が得られる．ここで C はマッチ後の式から得られる制約であるがこの場合にはサイズパラメータに関する制約はないため $C = \top$ となる．したがって，case 式から得られる制約は $\top \wedge (\forall p, q.(n = 1 + p + q \to \top))$ である．すなわち，この関数定義が適切であることを検査するためには，$\forall n.n > 0 \to (\top \wedge (\forall p, q.(n = 1 + p + q \to \top)))$ を検査すれば良い．実際にこの制約式は真であるため，この関数定義は適切である．

プログラム 4 は二つのヒープから新しいヒープを作る関数の定義である．where 節にて，関数呼び出しの際に必要な制約が記述されている．この場合，C を関数定義の右辺から得られる制約とすると，$\forall n, m.r = 1 + n + m \to C$ の真偽を判定することにより，この関数定義の妥当性を判定する．

3.1.4 再帰関数定義

制限のない再帰関数の定義を許す場合，停止性の保証されないプログラムの記述によりノード更新の停止性が保証できなくなってしまう．そのため，本研究で導入する再帰関数定義には制限が加えられている．再帰関数は引数に再帰的なデータ型を取り，かつ再帰呼び出しの際にその引数が構造的に単調減少することが明らかである場合にのみ定義が可能である．

再帰関数定義について例を挙げる．プログラム 5 はヒープ構造中の値の合計を求める再帰関数の定義である．sumHeap 関数内では，自身を再帰的に呼び出しているが，その引数には a と b が渡されている．a と b は元々の仮引数である h から分解されたものであるため，構造的に単調減少していることは明らかである．実際の検査の際には，a と b がそれぞれ型 Heap #p と型 Heap #q に割り当てられた場合，サイズパラメータの制約として $n > p \wedge n > q$ を追加すればよい．

3.1.5 ノード定義

プログラム 6 はヒープを保持するノードである．ノード h は型 Heap #21 を持つノードとして定義されている．10 個の値を保持するために必要なオブジェクト数は E と T を合わせて 21 個である．なお，ノード定義の際に使用できるサイズパラメータは定数パラメータのみである．

1 行目で使用されている cast オペレータはサイズパラメータを変換 (縮小) するための機能である．実行時のオブジェクトサイズを元にサイズを小さくできるならば新たに別のサイズパラメータを持った変数に束縛し後続の式を評価する，できないならば otherwise 節の式が評価される．この

```
53 ## ヒープ中の値の合計: forall #n, (Heap #n) -> Int
54 recfun sumHeap(h: Heap #n): Int = case h of
55   | E -> 0
56   | T(r, x, a, b) -> x + sumHeap(a) + sumHeap(b)
```

プログラム 5. ヒープ中の値の合計（再帰関数定義の例）

```
58 ## ヒープを保持する内部ノード
59 node [E #21] h: Heap #21 = cast h@last of            ## 直前値のサイズを変換する
60   | hl: #19    -> insert(x, hl)                      ## 9要素以下
61   | otherwise -> if x <= findMin(h@last) then h@last ## サイズ変換失敗 (10要素格納済み)
62                   else insert(x, deleteMin(h@last)) ## 最小値を削除して値を挿入
```

プログラム 6. ヒープを保持する内部ノード（ノード定義の例）

プログラムでは，cast オペレータにより直前時刻のヒープの空き容量を確認し，単純に値を追加するか最小値を削除したのちに値を追加するのかを決定している．

ノードの定義の妥当性も関数定義と同じように単純な型検査をしたのちにサイズパラメータに関して検査を行う．

3.1.6 実行時に必要なメモリ量の静的な決定

ノード更新の際に必要なメモリ領域はスタック領域とヒープ領域である．スタック領域は関数の引数やローカル変数 (ヒープ領域を差すポインタ) が配置される．ヒープ領域は生成されたオブジェクトが配置される．本研究において我々が決定するのは，あるノードの更新処理に必要なスタック領域とヒープ領域の大きさである．

ノードの更新中にはガベージコレクションは動作しないものとし，オブジェクトは全てヒープ領域に確保されるものとすると，スタック領域の必要量は関数呼び出しのネスト回数に，ヒープ領域の必要量はオブジェクトの生成回数に依存する．

case 式を使用していない場合，ノード定義ではサイズパラメータが定数であることを考慮すると，呼び出される関数のサイズパラメータを具体的に決定することができる．この時，一階の関数のみが定義可能であり，再帰関数については構造的な単調減少から最大呼び出し回数を見積もることが可能であるため，ノード更新に必要な関数呼び出しの回数や生成されるオブジェクトの個数は過大近似可能である．

case 式を使用していた場合，case 式によって分解されたあとのオブジェクトはサイズパラメータを具体的に表すことができない．例えばプログラム 5 の 4 行目では a と b は新たに導入されるサイズパラメータ#p，#q を用いて，Heap #p，Heap #q の型を持つ．サイズパラメータ#p，#q は制約 $n = 1 + p + q$ をもつが，#n に具体的な値が代入されたとしても#p，#q を具体的に定めることは一般にはできない．そこで，制約を満たす組み合わせを網羅的に設定し，それぞれの場合で関数呼び出しの回数や生成されるオブジェクトの個数を計算する．これにより得られた最大値を必要数として実行時に必要なリソース量を過大近似する．

3.2 考察

Emfrp には複合的なデータを扱う手段として代数的データ型が提供されている．各コンストラクタでは要素数が定義時に固定され追加や削除は行うことができない．Emfrp では配列が提供されておらず，添字を用いたデータ列へのアクセスは行えない．それゆえ，動的に要素数が変化するようなデータ列の取り扱いを Emfrp で記述する場合，代数的データ型を用いて要素数毎にコンストラクタを作成するか，タプルと Option 型を用いた表現になる．しかし，そのようなプログラムは動作の意味と実際の実装が乖離し，保守性の低い煩雑なプログラムになることが多い．本章で説明に用いたヒープ木も動的にデータの追加や削除が行われる例であり，例えば insert を記述するにはその時点での要素数や挿入位置を全て分岐によって記述する必要がある．

Emfrp では再帰関数の定義が禁止されているため，同じ処理を繰り返す場合には同様の動作を繰り返し記述する必要があった．提案手法により，制限付きであるが再帰関数が導入されたことで煩雑な繰り返しの記述は必要なくなり，ヒープ木への挿入のような構造に沿った処理を直観的で保守性の高いコードとして記述可能になったと言える．

4 FRP 言語への導入の定式化

Emfrp への再帰データ型の導入を定式化した．モジュールとノードの実行モデルは Emfrp と同様であると仮定する．つまり，ノードの依存関係に循環はなく，依存関係によるトポロジカル順にノードの更新処理が行われる．以下では一つのノードの更新処理についての形式化を行う．

4.1 構文

図 1 に構文を示す．モジュール内では関数定義，再帰関数定義，ノード定義，再帰データ型の定義を行うことができる．関数定義の際に引数や結果の型がサイズパラメータを含む場合，それは変数サイズパラメータであると制限する．関数定義の引数や結果の型で使用された変数サイズパラメータを定義式中で用いることができる．ノード定義式中では変数サイズパラメータは用いることができず，定数サイズパラメータのみを使用できる．

定数 c^{τ} は数値，ブール値，ユニット値が用意されている，二項演算子 op は四則演算や比較のような基本型同士の演算を想定している．`case` 式は再帰データ型の分解を行う構文である．ある型についての全てのコンストラクタが列挙されていると仮定する．`as` 式は再帰データ型のサイズパラメータの拡張に用いられ，より大きなサイズを持つ同じ型への変換を行う，`cast` 式は再帰データ型のサイズパラメータの縮小に用いられる．これは実行時にオブジェクトが持つ実際のサイズと与えられた定数サイズパラメータを比較し，その結果によって処理を分岐させる式である．n，n`@last` はそれぞれノード参照，直前値参照である．これらの式はノード定義式でのみ出現し，関数定義では出現しない．

コンストラクタ適用の際にサイズパラメータを設定できる．サイズパラメータは定数，変数とそれらの加減算により定義される．また，変数サイズパラメータは 0 よりも大きな値を取るとする．関数定義では `where` 節を使用して出力のサイズパラメータに対して等式制約をかけることができる．

4.2 型システム

図 2 に型の定義を示す．基本型は `Unit` 型，`Int` 型，`Bool` 型である．再帰データ型は常にサイズパラメータを持つ．基本型と再帰データ型を合わせてデータ型と定義する．プログラム中の変数やノードは全てデータ型 τ を持つ．関数は関数型 τ_f で定義された型を持つ．$\#\vec{k}$ は sc や引数，結果の型の中のサイズ変数の全称量化を表す．関数型中に自由なサイズ変数は存在しないものとする．関数型の定義から分かるように，本研究で提案する言語は一階の関数しか定義することはできない．

コンストラクタは関数型を持つ．再帰データ型宣言 ρ のコンストラクタの引数に現れる ρ には自動的にフレッシュなサイズパラメータが割り当てられ，それぞれのサイズパラメータの和に 1 を足したサイズが結果となるような制約がコンストラクタの型の制約となる．例えば，二分木は `rectype Tree = Leaf(Int) | Node(Tree, Tree)` と定義され，コンストラクタ `Leaf`，`Node` はそれぞれ関数型 $\forall$`#p.(Int)` $\to$ `Tree #p` と $\forall$`#p,#q.(#r = #1 + #p + #q) => (Tree #p, Tree #q)` $\to$ `Tree #r` を持つ．

図 3 に式の型付け規則を示す．提案言語の型付け規則では，型の検査とサイズパラメータに関する制約の抽出を同時に行う．型付け規則 $\Gamma \vdash e : \tau \mid c$ は「環境 Γ のもとで式 e は型 τ を持ち，サイズパラメータ制約 c をもつ」と読む．

Γ は型環境を表し変数，関数名と型のペアの列である，D はデータ型環境を表しコンストラクタ名と関数型のペアの列である．再帰データ型のコンストラクタは関数とみなされ，D に追加される．

$\simeq$ はサイズパラメータを除いた型の同値を表す関係である，右辺と左辺が同じ基本型であるとき，または右辺と左辺がサイズパラメータを除いて同じ再帰データ型を持つ時に関係が成り立つ．

定義

d	::=	recfun $f(x{:}\tau,\ldots,x{:}\tau){:}\tau$	
		$[\texttt{where}\ (sc)]$ = e	(再帰関数)
	$\mid$	node init$[e]$ $n{:}\tau$ = e	(ノード)
	$\mid$	rectype ρ = $\{C_i(\tau_{i1},\ldots,\tau_{il_i})\}_i$	(再帰データ型)

式

e	::=	c^τ	(定数)
	$\mid$	x	(変数)
	$\mid$	$C\ s\ (e,\ldots,e)$	(コンストラクタ)
	$\mid$	$e\ op^{(\tau_1,\tau_2)\to\tau}\ e$	(演算子適用)
	$\mid$	$f(e,\ldots,e)$	(関数適用)
	$\mid$	let x = e in e	(変数定義)
	$\mid$	if e then e else e	(if 式)
	$\mid$	case e of $\{C_i(x_{i1},\ldots,x_{il_i})$ -> $e_i\}_i$	(分解)
	$\mid$	e as s	(拡張変換)
	$\mid$	cast e of x : #c -> e	
		| otherwise -> e	(縮小変換)
	$\mid$	n	(ノード参照)
	$\mid$	n@last	(直前値参照)

サイズパラメータ

s	:=	#c	(定数パラメータ)
	$\mid$	#k	(変数パラメータ)
	$\mid$	s + s	(加算)
	$\mid$	s - s	(減算)

サイズパラメータ制約式

sc	::=	#k = s

図 1. 構文

補助関数 $\mathrm{sp}(\tau)$ は型からサイズパラメータへの写像であり，以下のように定義される．入力がサイズパラメータを持つ時そのパラメータを出力し，持たない時は#0 を出力する．

$$\mathrm{sp}(\tau) = \begin{cases} s & (\tau = \rho\ s) \\ \#0 & (\text{otherwise}) \end{cases}$$

関係 $\sqsubseteq$ はサイズパラメータによって全称量化された型のインスタンス化を表し，フレッシュな変数サイズパラメータの割り当てと置換が行われる，

通常の関数定義ではこの型付け規則に従って検査をし，得られた制約が充足可能かどうかを判定する．提案言語の型システムはサイズパラメータ以外は一般的な単相の型システムであるため，型検査は決定可能である．また，規則から得られる制約は整数の加減算，比較を含む一階の述語論理で十分に表される．これはプレスバーガー算術として知られた体系の範疇であり，その充足問題は決定可能である．従って関数定義の妥当性検査は決定可能である．

isRecursiveCall(f) は再帰関数呼び出しを区別するための述語である．関数 f の型検査中に同じ f の呼び出し式を検査する際に真になる．再帰関数定義の妥当性検査では通常の関数呼び出しの型付け規則 T-APP での c の制約に実引数のサイズパラメータの合計が仮引数のサイズパラメータの合計よりも小さいという制約を加えて制約の充足可能性を判定する必要がある．規則 T-RECAPP では，関数の再帰呼び出しを検出しこの制約を追加している．

型

$$
\begin{array}{lll}
\tau & ::= & \\
 & \mid \ \texttt{Unit} \ \mid \ \texttt{Int} \ \mid \ \texttt{Bool} & \text{(基本型)} \\
 & \mid \ \rho\, s & \text{(再帰データ型)} \\
\tau_f & ::= & \forall \overline{\#k}\,.\, sc \Rightarrow (\tau, \ldots, \tau) \to \tau \quad \text{(関数型)}
\end{array}
$$

図 2. 型

関数定義の妥当性を検査する方法を述べる．sc を関数定義の際の出力パラメータ制約，C を定義の右辺から得られた制約，$\overline{\#k}$ を sc と C に出現する自由変数の列とすると，$\forall\, \overline{\#k}.(\overline{\#k > 0} \wedge sc) \to C$ が充足可能であるかを検査する．sc は関数の入力と結果のサイズの関係を規定する制約である．この制約とサイズパラメータは正の値をとるという条件の下で式から抽出される制約が満たされるかどうかを検査している．以下のように規則として記述できる．

$$
\frac{\begin{array}{ll}\overline{\#k} = \mathrm{FV}(sc) & \Gamma, (f : \forall \overline{\#k}.sc \Rightarrow (\tau_1, \ldots, \tau_n) \to \tau), (x_1 : \tau_1), \ldots, (x_n : \tau_n) \vdash e : \tau' \mid c \\ \tau \simeq \tau' & \overline{\#k'} = \mathrm{FV}(c) \qquad c' = \{\forall \overline{\#k'}.\overline{\#k' > 0} \wedge sc \to c \wedge \mathrm{sp}(\tau) = \mathrm{sp}(\tau')\}\end{array}}{\Gamma \vdash \texttt{recfun}\ f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau\ \texttt{where}\ (sc) = e \mid c'}
\quad \text{(T-RECFUN)}
$$

ノード定義の妥当性は右辺の式を型付けし制約を検査する．以下にその規則を示す．ノード定義中にはサイズパラメータの出現は定数のみが許されていることに注意されたい．

$$
\frac{\Gamma \vdash e' : \tau \mid c' \qquad \Gamma, n : \tau \vdash e : \tau \mid c \qquad \overline{\#k} = \mathrm{FV}(c) \cup \mathrm{FV}(c') \qquad c'' = \{forall \overline{\#k}.\overline{\#k > 0} \to c \wedge c'\}}{\Gamma \vdash \texttt{node init[}e'\texttt{]}\ n{:}\tau = e \mid c''}
\quad \text{(T-DEFNODE)}
$$

4.3 操作的意味論

ノードの更新は定義式を実行することで行われる．図 4 に式の操作的意味論を示す．

規則 $E^s \mid H^t \vdash e \Downarrow_u l, H''^{t'}$ は「空き容量 s の実行環境 E，空き容量 t のヒープ領域 H，関数呼び出し可能回数 u の下で式 e は位置 l と新しい空き容量 t' のヒープ領域 H' へと評価される．ただし $l \in H'$ である.」と読む．

値は $v ::= () \mid n \mid True \mid False \mid C[k](l, \ldots, l)$ と定義される．ユニット値，数値，ブール値に加え，再帰データ型を表す値がある．C はコンストラクタ，k は「実行時に自身に自身と同じ構造がいくつ含まれるか」を表すカウンタ，l はヒープ領域の位置である．

実行環境 E^s は定義域の大きさが s に制限された，変数名 x から位置 l への対応を表す．これはあらかじめ固定の容量の定められた CPU のスタック領域に確保されたローカル変数のモデルである．ヒープ H^t は定義域の大きさが t に制限された，位置 l から値 v への対応を表す．これはサイズ制限のあるヒープ領域のモデルである．

規則中の N，L はそれぞれノードの現在値と直前値の位置を表し，ノード名と位置のペアの列である．F は関数定義を表し，関数名と引数名，定義式の組の列である．ノード数や関数の数は実行時に変動しないため，これら領域の大きさは静的に定めることができる．

再帰データ型のオブジェクトは実行時に実際のサイズ情報を値として持つ．E-CTOR 規則はコンストラクタ適用を表す．コンストラクタが適用される際には，引数に含まれる自身と同じ型のオブジェクトのサイズ情報の総和に 1 を足したサイズを設定している．これにより実行時の正確なサイ

$$\frac{}{\Gamma \vdash c^{\tau} : \tau \mid \top}\ \text{(T-CONST)} \qquad \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau \mid \top}\ \text{(T-VAR)} \qquad \frac{\tau = \Gamma(n)}{\Gamma \vdash n : \tau \mid \top}\ \text{(T-NODE)} \qquad \frac{\tau = \Gamma(n)}{\Gamma \vdash n@\text{last} : \tau \mid \top}\ \text{(T-ATLAST)}$$

$$\frac{\begin{array}{c} sc \Rightarrow (\tau'_1, \ldots, \tau'_n) \to \tau \sqsubseteq D(C) \\ \{\Gamma \vdash e_i : \tau_i \mid c_i\}_{i \in 1 \ldots n} \qquad \{\tau_i \simeq \tau'_i\}_{i \in 1 \ldots n} \qquad c = \{sc \wedge s = \mathrm{sp}(\tau) \wedge \bigwedge_{i \in 1 \ldots n} \mathrm{sp}(\tau_i) = \mathrm{sp}(\tau'_i) \wedge \bigwedge_{i \in 1 \ldots n} c_i\} \end{array}}{\Gamma \vdash C\ s\ (e_1, \ldots, e_n) : \tau \mid c}\ \text{(T-CTOR)}$$

$$\frac{\begin{array}{c} \neg\ \mathrm{isRecursiveCall}(f) \qquad sc \Rightarrow (\tau'_1, \ldots, \tau'_n) \to \tau \sqsubseteq \Gamma(f) \\ \{\Gamma \vdash e_i : \tau_i \mid c_i\}_{i \in 1 \ldots n} \qquad \{\tau_i \simeq \tau'_i\}_{i \in 1 \ldots n} \qquad c = \{sc \wedge \bigwedge_{i \in 1 \ldots n} \mathrm{sp}(\tau_i) = \mathrm{sp}(\tau'_i) \wedge \bigwedge_{i \in 1 \ldots n} c_i\} \end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau \mid c}\ \text{(T-APP)}$$

$$\frac{\begin{array}{c} \mathrm{isRecursiveCall}(f) \qquad sc \Rightarrow (\tau'_1, \ldots, \tau'_n) \to \tau \sqsubseteq \Gamma(f) \qquad \forall \overline{\#k}.sc^d \Rightarrow (\tau^d_1, \ldots, \tau^d_n) \to \tau^d = \Gamma(f) \\ \{\Gamma \vdash e_i : \tau_i \mid c_i\}_{i \in 1 \ldots n} \qquad \{\tau_i \simeq \tau'_i\}_{i \in 1 \ldots n} \\ c = \{sc \wedge \bigwedge_{i \in 1 \ldots n} \mathrm{sp}(\tau_i) = \mathrm{sp}(\tau'_i) \wedge \bigwedge_{i \in 1 \ldots n} c_i \wedge (\sum_{i \in 1 \ldots n} \mathrm{sp}(\tau_i) < \sum_{i \in 1 \ldots n} \mathrm{sp}(\tau^d_i))\} \end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau \mid c}\ \text{(T-RECAPP)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid c_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid c_2}{\Gamma \vdash e_1\ op^{(\tau_1, \tau_2) \to \tau}\ e_2 : \tau \mid c_1 \wedge c_2}\ \text{(T-OP)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \mid c_1 \qquad \Gamma, (x : \tau_1) \vdash e_2 : \tau_2 \mid c_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2 \mid c_1 \wedge c_2}\ \text{(T-LET)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \texttt{Bool} \mid c_1 \qquad \Gamma \vdash e_2 : \tau \mid c_2 \qquad \Gamma \vdash e_3 : \tau' \mid c_3 \\ \tau \simeq \tau' \qquad c = \{\mathrm{sp}(\tau) = \mathrm{sp}(\tau') \wedge c_1 \wedge c_2 \wedge c_3\} \end{array}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid c}\ \text{(T-IF)} \qquad \frac{\Gamma \vdash e : \rho\ s' \mid c}{\Gamma \vdash e\ \texttt{as}\ s : \rho\ s \mid c \wedge s \geq s'}\ \text{(T-AS)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : \tau \mid c_0 \qquad \left\{ \begin{array}{c} sc_i \Rightarrow (\tau_{i1}, \ldots, \tau_{il_i}) \to \tau_i \sqsubseteq D(C_i) \\ \tau \simeq \tau_i \qquad \Gamma, (x_{i1} : \tau_{i1}), \ldots, (x_{il_i} : \tau_{il_i}) \vdash e_i : \tau'_i \mid c_i \\ c'_i = \{(\mathrm{sp}(\tau) = \mathrm{sp}(\tau_i) \wedge sc_i) \to c_i\} \end{array} \right\}_{i \in 1 \ldots n} \qquad \left\{\ \tau'_1 \simeq \tau'_i\ \right\}_{i \in 1 \ldots n} \\ c = \{c_0 \wedge \bigwedge_{i \in 1 \ldots n} c'_i \wedge \bigwedge_{i \in 1 \ldots n} \mathrm{sp}(\tau'_1) = \mathrm{sp}(\tau'_i)\} \end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{C_i(x_{i1}, \ldots, x_{il_i})\ \texttt{->}\ e_i\}_i : \tau'_1 \mid c}\ \text{(T-CASE)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \rho\ s' \mid c_1 \qquad \Gamma, (x : \rho\ \#c) \vdash e_2 : \tau \mid c_2 \qquad \Gamma \vdash e_3 : \tau' \mid c_3 \\ \tau \simeq \tau' \qquad c = \{\mathrm{sp}(\tau) = \mathrm{sp}(\tau') \wedge c_1 \wedge c_2 \wedge c_3\} \end{array}}{\Gamma \vdash \texttt{cast}\ e_1\ \texttt{of}\ x : \#c\ \texttt{->}\ e_2\ \texttt{|}\ \texttt{otherwise}\ \texttt{->}\ e_3 : \tau \mid c}\ \text{(T-CAST)}$$

図 3. 式の型付け規則

ズを計算することができる．規則中では以下の補助関数 isSameType を用いて，コンストラクタが同じ型に由来したものであるかを検査している．

$$\begin{aligned}\mathrm{isSameType}(C, C') \quad = \quad & (\forall \overline{\#k} \,.\, sc \Rightarrow (\overline{\tau_{args}}) \to \tau = D(C)) \\ \wedge \quad & (\forall \overline{\#k'} \,.\, sc' \Rightarrow (\overline{\tau'_{args}}) \to \tau' = D(C')) \\ \wedge \quad & \tau \simeq \tau'\end{aligned}$$

E-CAST-SUCCESS 規則と E-CAST-FAIL 規則はそれぞれ cast オペレータの動作を記述した規則である．実行時のサイズ情報と定数サイズパラメータを比較し処理を分岐している．

4.4 メモリ使用量の過大近似

実行時に必要なメモリ量の過大近似とは，意味論のパラメータ s，t，u が実行中に負の値にならないような十分に大きな値を決定することである．図 5 に示すアルゴリズムを用いることで十分に大きなパラメータ s，t，u が計算できる．

近似アルゴリズムは関数 $\mathcal{M}_{\Delta,\Sigma}[\![e]\!]$ と関数 $\mathcal{A}^{P_0}_{\Delta',\Sigma}[\![arm]\!]$ によって，相互再帰的に定義される．それぞれの関数は正常に型付けされ制約の充足が確認された式や case 式の 1 つの節を入力とし，4 つの整数値 (p, s, t, u) を出力する．p は入力された式の型が持つサイズパラメータの具体値である．ただし型が基本型であるときは 0 である．s，t，u はそれぞれ入力された式を評価する際に必要な変数スタックの大きさ，ヒープの使用量，コールスタックの大きさ (関数の呼び出し回数) である．あるノードの更新の際にはノード定義式を起点とし，関数 $\mathcal{M}$ によって得られたパラメータ s，t，u を用いることで操作的意味論の実行を行うことができる．Δ は関数の引数や結果の型に現れる変数サイズパラメータと，その関数の呼び出し時に設定された具体的なサイズ (具体値) のペアの列である．Σ は関数の仮引数や定義された変数とその変数の型に含まれるサイズパラメータの具体値のペアである．アルゴリズム中の $eval_\Delta(s)$ は構文で表されたサイズを Δ の環境で評価し，具体的な数値 (具体値) を計算する補助関数である．補助関数 EachMax は与えられた条件下で計算した際の 4 つ組それぞれの最大値を出力するものとする．4 つ組の最大値は同時に出力される組ではないということに注意されたい．

アルゴリズムの概要を述べる．ノードの定義式中に現れる定数サイズパラメータを具体値と呼ぶ．ノード定義式や初期化式内では定数のサイズパラメータのみが使用可能である．そのため，ノード定義式と初期化式を起点とし，サイズパラメータの具体値を伝搬させながら構文木を辿って実行時に行われるであろう関数呼び出しをシミュレートすることで，実行時に必要なリソース量を計算する．ノード更新処理中にガベージコレクションなどのヒープ領域を解放するような処理が発生しないと仮定して，必要リソースの過大近似を行う．入力として与えられる式は正常に型付けされ，制約が検査されたものであるため関数の再帰的な呼び出しが停止することが事前に示されていることから，このアルゴリズムの探索も停止することがわかる．アルゴリズム中の最も重要な部分は case 式に対する近似処理である．与えられた再帰データ型の式のサイズを分解後の変数にどれだけ割り振った時が最もリソースを消費するかを網羅的に探索し，必要リソースの最大を見積もる．

プログラム全体としてはノード更新に必要なリソース量をノードごとに計算し，その最大値をそれぞれ事前に確保すればよい．また，ノードの現在値と直前値を格納するための領域をあらかじめ確保して，ノードの更新処理の最後にその領域にデータを格納し，ヒープやスタック領域をリセットすることで一定のメモリリソースでプログラムの実行を続けることが可能である．

5 関連研究

本研究の提案手法は SizedTypes[10] に大きく影響を受けている．SizedTypes は組込みリアクティブシステムに関する性質を検証するために設計された型システムである．型に対してサイズ情報を

$$\frac{H' = H, l \mapsto c^\tau \qquad l \notin H}{E^s \mid H^t \vdash c^\tau \Downarrow_u l, H'^{t-1}} \text{(E-CONST)} \qquad \frac{l = E(x)}{E^s \mid H^t \vdash x \Downarrow_u l, H^t} \text{(E-VAR)}$$

$$\frac{l = N(n)}{E^s \mid H^t \vdash n \Downarrow_u l, H^t} \text{(E-NODE)} \qquad \frac{l = L(n)}{E^s \mid H^t \vdash n\texttt{@last} \Downarrow_u l, H^t} \text{(E-ATLAST)}$$

$$\frac{\begin{array}{c} H_0 = H \qquad t_0 = t \qquad \left\{ \; E^s \mid H_{i-1}^{t_{i-1}} \vdash e_i \Downarrow_u l_i, H_i^{t_i} \; \right\}_{i \in 1 \ldots n} \\ k = 1 + \sum\{k_i \mid i \in 1 \ldots n, H_n(l_i) = C_i[k_i](l'_1, \ldots, l'_m), \text{isSameType}(C, C_i)\} \\ v = C[k](l_1, \ldots, l_n) \qquad t' = t_n - 1 \qquad H' = H_n, l \mapsto v \qquad l \notin H_n \end{array}}{E^s \mid H^t \vdash C\ s\ (e_1, \ldots, e_n) \Downarrow_u l, H'^{t'}} \text{(E-CTOR)}$$

$$\frac{\begin{array}{c} H_0 = H \qquad t_0 = t \qquad \left\{ \; E^s \mid H_{i-1}^{t_{i-1}} \vdash e_i \Downarrow_u l_i, H_i^{t_i} \; \right\}_{i \in 1 \ldots n} \\ ((x_1, \ldots, x_n), e) = F(f) \qquad (x_1 \mapsto l_1, \ldots, x_n \mapsto l_n)^{s-n} \mid H_n^{t_n} \vdash e \Downarrow_{u-1} l', H'^{t'} \end{array}}{E^s \mid H^t \vdash f(e_1, \ldots, e_n) \Downarrow_u l', H'^{t'}} \text{(E-APP)}$$

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e_1 \Downarrow_u l_1, H_1^{t_1} \qquad E^s \mid H_1^{t_1} \vdash e_2 \Downarrow_u l_2, H_2^{t_2} \\ v_1 = H_1(l_1) \qquad v_2 = H_2(l_2) \qquad v = op(v_1, v_2) \qquad H_3 = H_2, l \mapsto v \qquad l \notin H_2 \end{array}}{E^s \mid H^t \vdash e_1\ op^{(\tau_1, \tau_2) \to \tau}\ e_2 \Downarrow_u l, H_3^{t_2 - 1}} \text{(E-OP)}$$

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

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e_1 \Downarrow_u l_1, H_1^{t_1} \\ H_1(l_1) = \texttt{True} \qquad E^s \mid H_1^{t_1} \vdash e_2 \Downarrow_u l_2, H_2^{t_2} \end{array}}{E^s \mid H^t \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_2, H_2^{t_2}} \text{(E-IF-THEN)}$$

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e_1 \Downarrow_u l_1, H_1^{t_1} \\ H_1(l_1) = \texttt{False} \qquad E^s \mid H_1^{t_1} \vdash e_3 \Downarrow_u l_2, H_2^{t_2} \end{array}}{E^s \mid H^t \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_2, H_2^{t_2}} \text{(E-IF-ELSE)}$$

$$\frac{E^s \mid H^t \vdash e \Downarrow_u l, H_1^{t'}}{E^s \mid H^t \vdash e\ \texttt{as}\ s \Downarrow_u l, H_1^{t'}} \text{(E-AS)}$$

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e \Downarrow_u l_1, H_1^{t_1} \qquad C[k](l'_1, \ldots, l'_n) = H_1(l_1) \\ \exists! m. \left\{ \; C = C_m \qquad (E, x_{m1} \mapsto l'_1, \ldots, x_{ml_m} \mapsto l'_n)^{s-n} \mid H_1^{t_1} \vdash e_m \Downarrow_u l_2, H_2^{t_2} \; \right\} \end{array}}{E^s \mid H^t \vdash \texttt{case}\ e\ \texttt{of}\ \{C_i(x_{i1}, \ldots, x_{il_i})\ \texttt{->}\ e_i\}_i \Downarrow_u l_2, H_2^{t_2}} \text{(E-CASE)}$$

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e_1 \Downarrow_u l_1, H_1^{t_1} \\ C[k](l'_1, \ldots, l'_n) = H_1(l_1) \qquad k \leq c \qquad (E, x \mapsto l_1)^{s-1} \mid H_1^{t_1} \vdash e_2 \Downarrow_u l_2, H_2^{t_2} \end{array}}{E^s \mid H^t \vdash \texttt{cast}\ e_1\ \texttt{of}\ x : \texttt{\#}c\ \texttt{->}\ e_2\ \texttt{|}\ \texttt{otherwise}\ \texttt{->}\ e_3 \Downarrow_u l_2, H_2^{t_2}} \text{(E-CAST-SUCCESS)}$$

$$\frac{\begin{array}{c} E^s \mid H^t \vdash e_1 \Downarrow_u l_1, H_1^{t_1} \\ C[k](l'_1, \ldots, l'_n) = H_1(l_1) \qquad k > c \qquad E^s \mid H_1^{t_1} \vdash e_3 \Downarrow_u l_2, H_2^{t_2} \end{array}}{E^s \mid H^t \vdash \texttt{cast}\ e_1\ \texttt{of}\ x : \texttt{\#}c\ \texttt{->}\ e_2\ \texttt{|}\ \texttt{otherwise}\ \texttt{->}\ e_3 \Downarrow_u l_2, H_2^{t_2}} \text{(E-CAST-FAIL)}$$

図 4. 式の操作的意味論

$$
\begin{array}{lcl}
\mathcal{M}_{\Delta,\Sigma}[\![c^{\tau}]\!] & = & (0,0,1,0) \\
\mathcal{M}_{\Delta,\Sigma}[\![x]\!] & = & (\Sigma(x),0,0,0) \\
\mathcal{M}_{\Delta,\Sigma}[\![n]\!] & = & (\Sigma(n),0,0,0) \\
\mathcal{M}_{\Delta,\Sigma}[\![n\texttt{@last}]\!] & = & (\Sigma(n),0,0,0) \\
\mathcal{M}_{\Delta,\Sigma}[\![C\ s\ (e_1,\ldots,e_n)]\!] & = & (\mathrm{eval}_{\Delta}(s),\mathrm{Max}\{s_1,\ldots,s_n\},1+t_1+\ldots+t_n,\mathrm{Max}\{u_1,\ldots,u_n\}) \\
 & \textit{where} & \{(\_,s_i,t_i,u_i)=\mathcal{M}_{\Delta,\Sigma}[\![e_i]\!]\}_{i\in 1\ldots n} \\
\mathcal{M}_{\Delta,\Sigma}[\![f(e_1,\ldots,e_n)]\!] & = & (p',n+\mathrm{Max}\{s',s_1,\ldots,s_n\},t'+t_1+\ldots+t_n,\mathrm{Max}\{1+u',u_1,\ldots,u_n\}) \\
 & \textit{where} & \{(p_i,s_i,t_i,u_i)=\mathcal{M}_{\Delta,\Sigma}[\![e_i]\!]\}_{i\in 1\ldots n} \\
 & & \Delta' : f \text{ の引数と結果の型に含まれるサイズパラメータと} \\
 & & \quad \text{具体値のマッピング} \\
 & & \Sigma' : f \text{ の仮引数名と} \\
 & & \quad \text{実引数の型のサイズの具体値のマッピング} \\
 & & (p',s',t',u')=\mathcal{M}_{\Delta',\Sigma'}[\![f \text{ の定義式}]\!] \\
\mathcal{M}_{\Delta,\Sigma}[\![e_1\ op^{(\tau_1,\tau_2)\to\tau}\ e_2]\!] & = & (0,\mathrm{Max}\{s_1,s_2\},1+t_1+t_2,\mathrm{Max}\{u_1,u_2\}) \\
 & \textit{where} & (\_,s_1,t_1,u_1)=\mathcal{M}_{\Delta,\Sigma}[\![e_1]\!]\ ,\ (\_,s_2,t_2,u_2)=\mathcal{M}_{\Delta,\Sigma}[\![e_2]\!] \\
\mathcal{M}_{\Delta,\Sigma}[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] & = & (p_2,\mathrm{Max}\{s_1,1+s_2\},t_1+t_2,\mathrm{Max}\{u_1,u_2\}) \\
 & \textit{where} & (p_1,s_1,t_1,u_1)=\mathcal{M}_{\Delta,\Sigma}[\![e_1]\!]\ ,\ (p_2,s_2,t_2,u_2)=\mathcal{M}_{\Delta,(\Sigma,x:p_1)}[\![e_2]\!] \\
\mathcal{M}_{\Delta,\Sigma}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & (p,\mathrm{Max}\{s_1,s_2,s_3\},t_1+\mathrm{Max}\{t_2,t_3\},\mathrm{Max}\{u_1,u_2,u_3\}) \\
 & \textit{where} & (0,s_1,t_1,u_1)=\mathcal{M}_{\Delta,\Sigma}[\![e_1]\!] \\
 & & (p,s_2,t_2,u_2)=\mathcal{M}_{\Delta,\Sigma}[\![e_2]\!]\ ,\ (p,s_3,t_3,u_3)=\mathcal{M}_{\Delta,\Sigma}[\![e_3]\!] \\
\mathcal{M}_{\Delta,\Sigma}[\![e\ \texttt{as}\ s]\!] & = & (\mathrm{eval}_{\Delta}(s),s,t,u)\quad \textit{where}\ (\_,s,t,u)=\mathcal{M}_{\Delta,\Sigma}[\![e]\!] \\
\mathcal{M}_{\Delta,\Sigma}[\![\texttt{case}\ e\ \texttt{of}\ \{C_i(x_{i1},\ldots,x_{il_i})\ \texttt{->}\ e_i\}_{i\in 1\ldots n}]\!] & & \\
 & = & (p,\mathrm{Max}\{s_0,s_1+l_1,\ldots,s_n+l_n\}, \\
 & & \quad t_0+\mathrm{Max}\{t_1,\ldots,t_n\},\mathrm{Max}\{u_0,u_1,\ldots,u_n\}) \\
 & \textit{where} & (p_0,s_0,t_0,u_0)=\mathcal{M}_{\Delta,\Sigma}[\![e]\!] \\
 & & \{(p,s_i,t_i,u_i)=\mathcal{A}^{p_0}_{\Delta,\Sigma}[\![C_i(x_{i1},\ldots,x_{il_i})\ \texttt{->}\ e_i]\!]\}_{i\in 1\ldots n} \\
\mathcal{M}_{\Delta,\Sigma}[\![\texttt{cast}\ e_1\ \texttt{of}\ x:\#c\ \texttt{->}\ e_2\ \texttt{|}\ \texttt{otherwise}\ \texttt{->}\ e_3]\!] & & \\
 & = & (p,\mathrm{Max}\{s_1,1+s_2,s_3\},t_1+\mathrm{Max}\{t_2,t_3\},\mathrm{Max}\{u_1,u_2,u_3\}) \\
 & \textit{where} & (\_,s_1,t_1,u_1)=\mathcal{M}_{\Delta,\Sigma}[\![e_1]\!] \\
 & & (p,s_2,t_2,u_2)=\mathcal{M}_{\Delta,(\Sigma,x:c)}[\![e_2]\!]\ ,\ (p,s_3,t_3,u_3)=\mathcal{M}_{\Delta,\Sigma}[\![e_3]\!] \\
\mathcal{A}^{p_0}_{\Delta,\Sigma}[\![C(x_1,\ldots,x_l)\ \texttt{->}\ e]\!] & = & (p,s,t,u) \\
 & \textit{where} & \forall\overline{\#k}.sc \Rightarrow (\tau_1,\ldots,\tau_l)\to\tau = D(C) \\
 & & (p,s,t,u)=\mathrm{EachMax}\{\mathcal{M}_{\Delta,(\Sigma,x_1:q_1,\ldots,x_l:q_l)}[\![e]\!] \\
 & & \qquad\qquad \mid \textstyle\sum_{j\in 1\ldots l}\delta_{\tau_j,\tau}*w_j=p_0, \\
 & & \qquad\qquad j\in 1\ldots l, \\
 & & \qquad\qquad q_j=\delta_{\tau_j,\tau}*w_j+(1-\delta_{\tau_j,\tau})*\mathrm{eval}_{\Delta}(\mathrm{sp}(\tau_j))\} \\
 & & \delta_{\tau,\tau'}=\begin{cases}1 & (\tau\simeq\tau') \\ 0 & (\text{otherwise})\end{cases}
\end{array}
$$

図 5. 式の使用メモリ量過大近似アルゴリズム

付与し，サイズに上限や下限のある型を使用してプログラムを記述することでデッドロックの検出，個々のストリーム計算の停止性判定，一定メモリ量での動作保証などが解析できる．

提案手法では SizedTypes でのサイズに上限のある型を取り入れることで，Emfrp の特徴を残しながらも再帰データ型の使用を緩和した．SizedTypes との差分としては使用メモリ量の静的な過大近似を行う点が挙げられる．一定メモリ量での動作は組込みシステムにとって非常に有用な性質であるが，実行前にその上限がわかることでメモリに関するエラーが起きないことをより強く保証することができる．

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 FRP 言語として Juniper[8] 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 [7]．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．

Wan らによる RT-FRP[14] および E-FRP[15] はリアルタイムシステム向けの FRP 言語であり，プログラムが使用するリソース（項の大きさ）の上限を静的に検査することができる．E-FRP はマイクロコントローラ (PIC16C66) で動作する C プログラムへのコンパイラが実装されている．これらの言語では，プログラムが要求するリソース量全体の静的な検査はできるが，あらかじめデータ構造のサイズを規定する等はできない（コンパイルしないとわからない）．一方本研究での提案手法では，サイズパラメータを用いることによりデータ構造毎にきめ細かいリソース量の制御が可能である．

Hume[5] 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の記述によってプログラムを構成する．Hume は関数を持たない最も単純なサブセットから，再帰的な関数とデータ型をサポートするフルセットまでいくつかのレベルがあり，その中の FSM-Hume と呼ばれるレベルについては空間コストモデルが定義され，スタックやヒープの使用量の静的解析が可能になっている [6]．FSM-Hume は関数とデータ型における再帰と高階関数を許さないモデルであり，その点では Emfrp に近い．ただし Emfrp は連続的に変化する時変値をサポートする FRP 言語であり，離散イベントのみをサポートする Hume とは異なる．

6　まとめと今後の課題

小規模組込みシステム向け FRP 言語 Emfrp に対し再帰データ型と再帰関数定義の導入を行った．再帰データ型についてはサイズパラメータを設定し，実行時に必要なメモリ量の静的な決定を可能にした．再帰関数については，呼び出し時の引数の構造的な単調減少に注目し，停止性が保証できる場合に限り定義を許した．これらの拡張を取り入れた FRP 言語を新たに提案し，構文，型システム，操作的意味論，メモリ使用量の過大近似について説明した．

現状ではこれらの提案に厳密な証明はなされておらず，処理系も存在しない．

今後の課題としては型システムやメモリ使用量近似アルゴリズムの健全性を証明し理論的な保証を与えること，処理系の実装を行うことが挙げられる．処理系を用いてある程度の規模のプログラムに対するコンパイル時間計測や実行時の時間的，空間的オーバーヘッドの計測の実験を予定している．

謝辞　改稿にあたり査読者より有益な助言およびコメントをいただいた．ここに感謝する．本研究の一部は JSPS 科研費 18K11236 および 19K20245 の助成を受けている．

参考文献

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

[2] Evan Czaplicki. A farewell to FRP: Making signals unnecessary with the Elm architecture. http://elm-lang.org/blog/farewell-to-frp, May 2016.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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), 2020. 掲載予定.

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

[13] 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.

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

[15] 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.

[16] 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.

付録 A　MaxSum10 モジュール

```
## 入力されたデータの上位 10番目までの合計
module MaxSum10
in  x: Int    ## 入力値
out y: Int    ## 上位データの合計値
use Std

## 左偏ヒープ (再帰データ型定義)
rectype Heap = E(Unit)                  ## 末端
             | T(Int, Int, Heap, Heap) ## ノード (ランク , 値 , 左 , 右)

## 木のランクを取得する: forall #n. (Heap #n) -> Int
fun rank(e: Heap #n) : Int = case e of E -> 0 | T (r, x, a, b) -> r

## ヒープを構築する ( 値を 2つの木の上に追加する )
## forall #n,#m. (Int, Heap #n, Heap #m) -> Heap #r
fun make(x: Int, a: Heap #n, b: Heap #m) : Heap #r where (#r = 1 + #n + #m) =
    let ra = rank(a) in
    let rb = rank(b) in
    if ra > rb then T #r (rb+1, x, a, b) else T #r (ra+1, x, b, a)

## 木が空か判定: forall #n. (Heap #n) -> Bool
fun isEmpty(h: Heap #n): Bool = case h of E -> True | _ -> False

## 2つのヒープをマージする
## forall #n,#m,#r. (#r=#n+#m-1) => (Heap #n, Heap #m) -> Heap #r
recfun merge(h1: Heap #n, h2: Heap #m) : Heap #r where (#r = #n + #m - 1) =
  case h1 of
  | E -> h2 as #r
  | T(r1, x1, a1, b1) -> begin
      case h2 of
      | E -> h1 as #r
      | T(r2, x2, a2, b2) -> if x1 <= x2 then
            let u1 = merge (b1, h2) in make(x1, a1, u1)
          else
            let u2 = merge (h1, b2) in make(y, a2, u2)
    end

## 値をヒープに挿入
## forall #n, #m. (#r=#n+2) => (Int, Heap #n) -> Heap #r
fun insert (x: Int, h: Heap #n): Heap #r where (#r = #n + 2) =
  merge (T #3 (1, x, E #1, E #1), h)

## 最小値の取得: forall #n. (Heap #n) -> Int
fun findMin(h: Heap #n): Int = case h of
  | E -> 0
  | T (_, x, _, _) -> x

## 最小値の削除: forall #n,#r. (#r=#n-2) => (Heap #n) -> Heap #r
fun deleteMin(h: Heap #n): Heap #r where (#r=#n-2) = case h of
  | E -> E #r
  | T (_, x, a, b) -> merge (a, b)

## ヒープ中の値の合計: forall #n, (Heap #n) -> Int
recfun sumHeap(h: Heap #n): Int = case h of
  | E -> 0
  | T(r, x, a, b) -> x + sumHeap(a) + sumHeap(b)

## ヒープを保持する内部ノード
node [E #21] h: Heap #21 = cast h@last of
  | hl: #19   -> insert(x, hl)
  | otherwise -> if x <= findMin(h@last) then h@last
                 else insert(x, deleteMin(h@last))

## 合計値を計算する出力ノード
node y: Int = sumHeap(h)
```

プログラム **7.** MaxSum10 モジュール