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

東京工業大学 情報理工学院
横山陽彦 森口草介 渡部卓雄

研究概要

- 小規模組み込み向け関数リアクティブプログラミング言語Emfrpに対し，オブジェクトサイズを見積もるためのパラメータが型に付与された再帰的なデータ型と関数を導入した

- 貢献
  - Emfrpの機能的制限の緩和
    - 一般的な関数プログラミングに用いられる機構の提供
    - 典型的なデータ構造の自然な表現
  - サイズパラメータを付与した型システムの提案
  - 使用メモリ量の静的に決定するアルゴリズムの提案

関数リアクティブプログラミング(FRP)

- リアクティブシステム
  - ex.) 組み込みシステム・GUI・アニメーション
  - ポーリングや割り込みを多用して記述される
    - プログラムの見通しが悪くなる

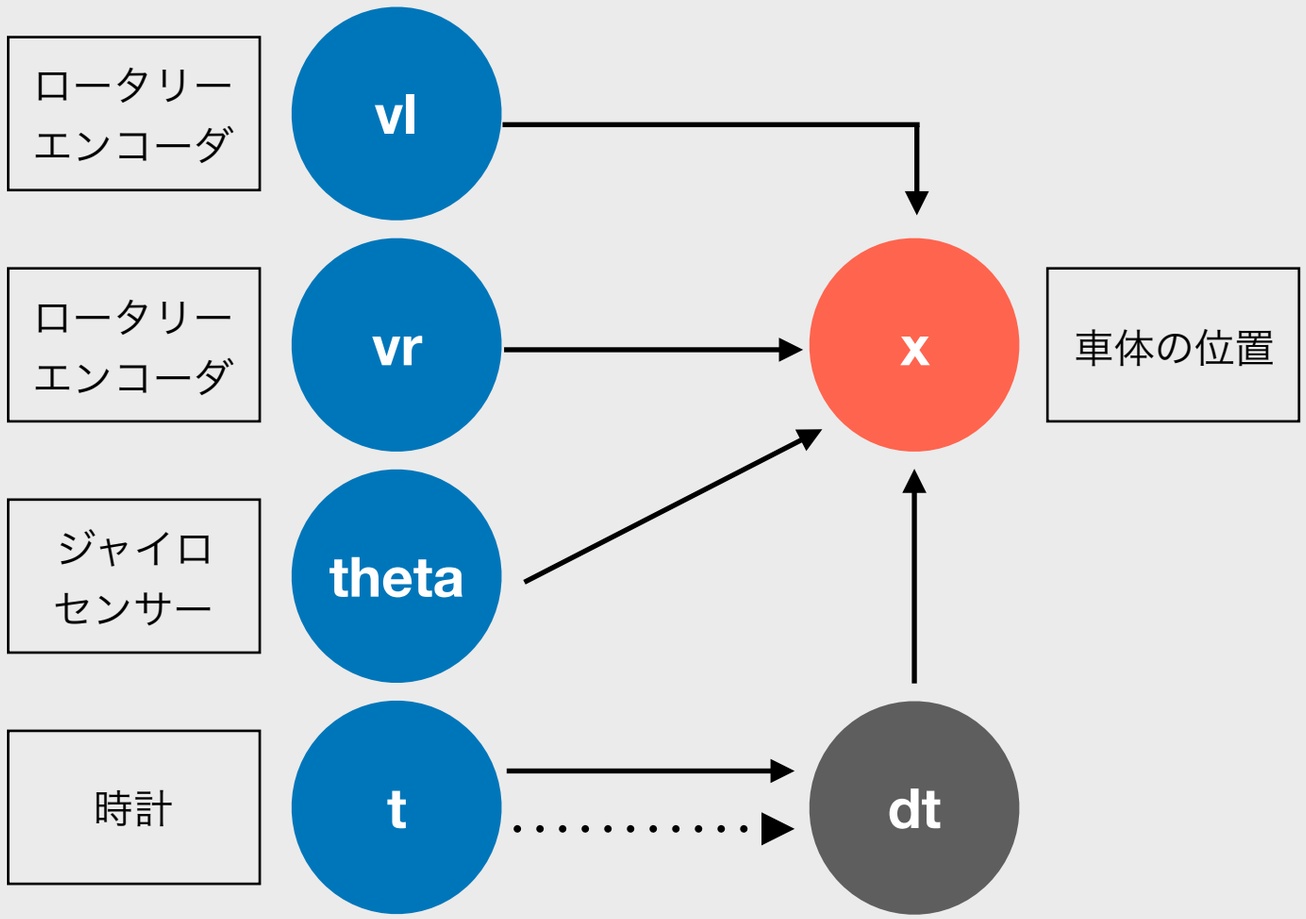
- 関数リアクティブプログラミング(FRP)
  - 時変値(time-varying value)
    - 時間とともに連続的，離散的に変化する値の抽象化
  - 時変値を組み合わせてリアクティブシステムを宣言的に記述

Emfrp [Sawada et al. 2016]

- 小規模組込みシステム向けFRP言語
  - 純粋・静的型付け
- マイクロコントローラ上で実行可能
  - 主記憶が数KB程度・CPU性能に制限がある
  - メモリ管理ユニットのない実行環境
  - ex.) AVR, ESP32, STM32
- 時間漏れ(time-leak)・空間漏れ(space-leak)がない
  - 時変値が第一級ではない
- 実行時のメモリ使用量を静的に決定可能
  - 関数やデータ型の再帰的な定義, 使用の禁止
  - 高階関数の定義, 使用の禁止

Emfrpプログラム

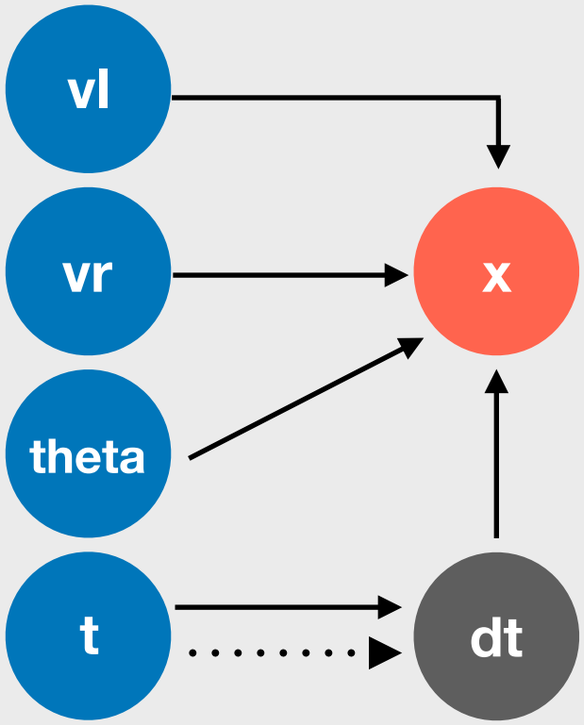
```
module Distance ## 2輪ロボットの位置(x座標)
in vl : Float, vr : Float, theta : Float,
   t(0)  : Int
out x : Float
use Std

node dt = (t - t@last) / 1000.0
node init[0.0] x =
  x@last + (vr + vl) * cos(theta) * dt / 2
```



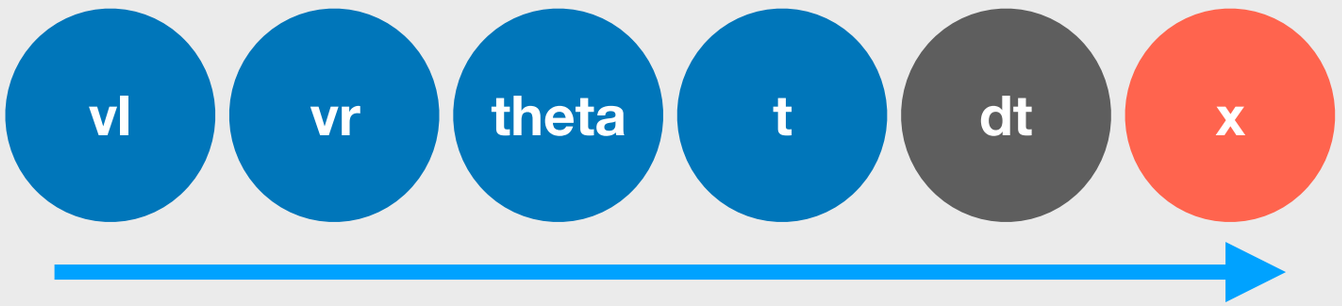
- モジュール単位でプログラムを記述
  - 入出力ノードの定義
  - 内部ノードの定義
- 時変値を**node**で定義
- 直前値を**x@last**で参照
  - プログラム起動時の直前値の初期値が必要
- ノードの依存関係はDAGとして表現
  - 実線は現在値の依存関係
  - 破線は直前値の依存関係

Emfrpの実行モデル

- 入力ノードの変化を定期的に伝搬させるpush型の更新処理
  - 実行開始時にノードの初期値を計算後，更新処理を繰り返す
  - 更新処理では直前値設定，入力ノード更新，ノード再計算を行う
  - 更新処理は現在値の依存関係のトポロジカル順序で行われる
  - OSによるスケジューリングは不要



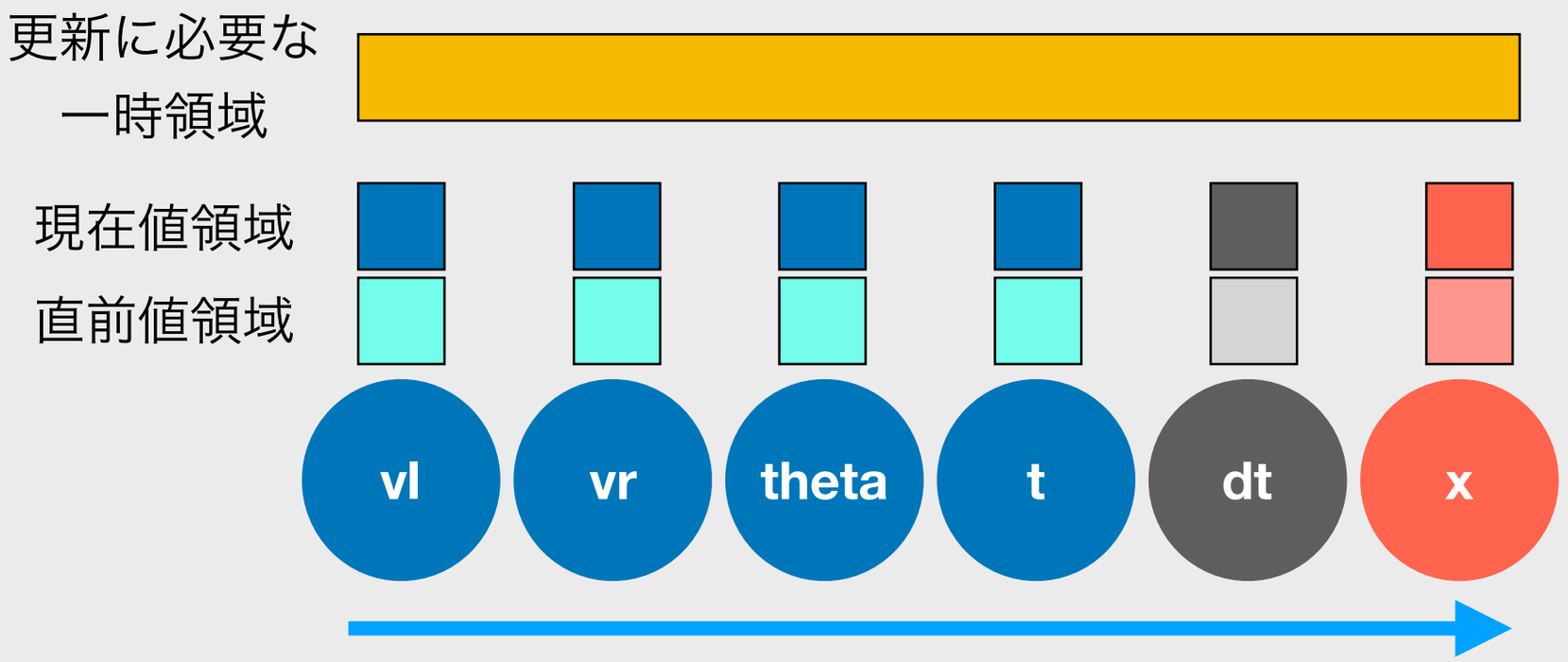
依存グラフ



ノード更新順序

Emfrpのメモリ管理

- 実行時に使用するメモリ量はコンパイル時に決定される
  - それぞれのノードの時変値(現在値), 直前値を保持する領域
  - ノードの更新処理に必要な一時的なメモリ領域
    - 1ノードずつ更新されるので, 最も多くメモリを要求するノードに合わせる
  - 不要なデータは更新処理毎にGCされる
- 実行時にメモリに関するエラーが起こらない



動機

- 問題点
  - Emfrpでは再帰的なデータ型，関数の定義ができない
    - リストや木などのデータ構造の自然な表現が困難
    - これらのデータ構造は小規模組込みシステムであっても有用
  - 単純な導入
    - 停止しない計算，メモリを無制限に使用する計算が記述可能

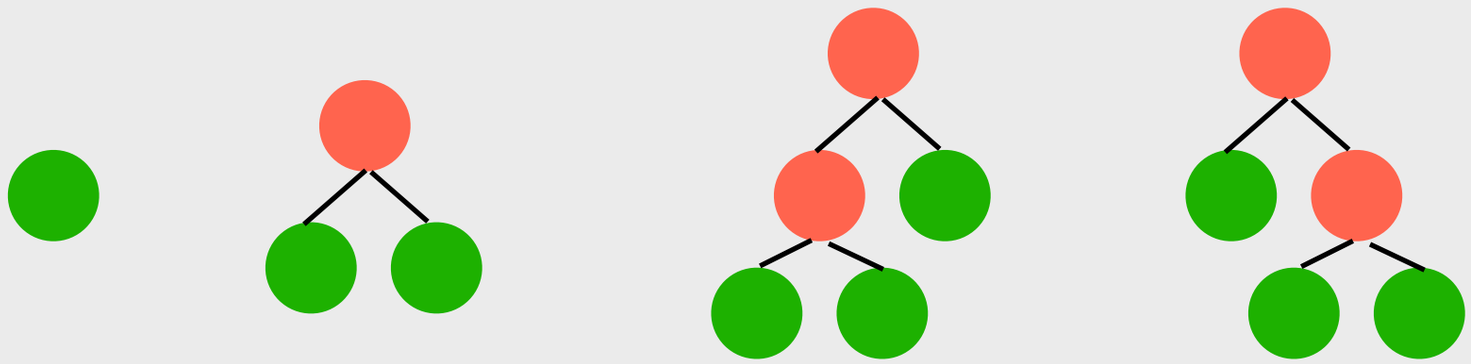
- Emfrpの持つ以下の性質を維持
  - メモリ使用量の決定
  - ノード更新処理の停止性の保証

提案手法

- 再帰的なデータ型に対して，サイズを見積もるためのパラメータを追加
  - 「そのオブジェクト内に自身と同じ構造が最大で何個出現するか」を型システムで管理
  - オブジェクトサイズの見積もり
- 原始再帰関数の導入
  - 停止性を保証した再帰呼び出し
- メモリ使用量の過大近似
  - サイズパラメータの網羅的な探索

サイズパラメータ

- 再帰データ型に付与されるオブジェクトサイズを見積もるためのパラメータ
  - 「そのオブジェクト内に自身と同じ構造が最大で何個出現するか」
  - サイズパラメータは定数(#0,#1,#2,...)，変数(#n,#m,...)とそれらの加減算が設定可能，サイズ変数は正の整数である
  - 例: **rectype** Tree = L | N(Tree, Tree)で定義された型Tree #5の値



- 関数は引数と結果のサイズパラメータに関する制約を型に含む
  - 制約は関数を呼び出す際の条件として扱われる
  - 例: リストのcons関数の型
    - `forall #n,#r.(#r=#n+1) => (Int, List #n) -> List #r`

例：ランキングモジュール

- 今までに入力された上位10個の値を保持，合計を出力
- ランキングはヒープ木(左偏ヒープ)で管理
  - 再帰データ型，再帰関数を使用
  - ヒープ木を表すノードを定義

```
## 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

...
```

モジュール定義

データ型定義(左偏ヒープ)

```
## 左偏ヒープ
rectype Heap = E(Unit) ## 末端
             | T(Int, Int, Heap, Heap) ## ノード（ランク，値，左，右）
```

```
• E: forall #n.(Unit) -> Heap #n
• T: forall #n,#m,#r.(#r=#n+#m+1) => (Int, Int, Heap #n, Heap #m) -> Heap #r
```

Heapのコンストラクタの型

- **rectype**で再帰データ型を定義
  - 定義中にはサイズパラメータは出現しない
- コンストラクタは関数として扱われる
  - 自動的にサイズパラメータが付与される
  - 「引数のサイズ合計に1足したものが結果のサイズ」という制約が課される

make関数の型・サイズ制約

```
## 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.(#r=1+#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)
```

$$\begin{aligned}
&\forall r, n, m, r_1, n_1, m_1, r_2, n_2, m_2.\ (r = 1 + n + m) \rightarrow \\
&\qquad (r_1 = r \wedge n_1 = n \wedge m_1 = m \wedge r_1 = n_1 + m_1 + 1) \\
&\wedge \quad (r_2 = r \wedge n_2 = m \wedge m_2 = n \wedge r_2 = n_2 + m_2 + 1)
\end{aligned}$$

make関数から得られるサイズ制約

- 関数定義では型検査，サイズに関する制約検査を行う
- **where**節の制約を前提として定義式から得られる制約を検査する
- コンストラクタ，関数呼び出しの際にフレッシュなサイズ変数を生成，制約を追加する

sumHeap関数の型・サイズ制約

```
## 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)
```

$$\forall n, p, q.\ \top \rightarrow ((n = p + q + 1) \rightarrow p{<}n \wedge q{<}n)$$

sumHeap関数から得られるサイズ制約

- **case**式では「入力サイズと一致する」フレッシュなサイズ変数を導入する (例のサイズ変数p,q)
- 関数の再帰呼び出しの際には, 「実引数のサイズが減少している」という制約を追加する
- **where**句が省略された場合は関数定義の制約式の前件は真として扱う

ノード定義とcast式

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

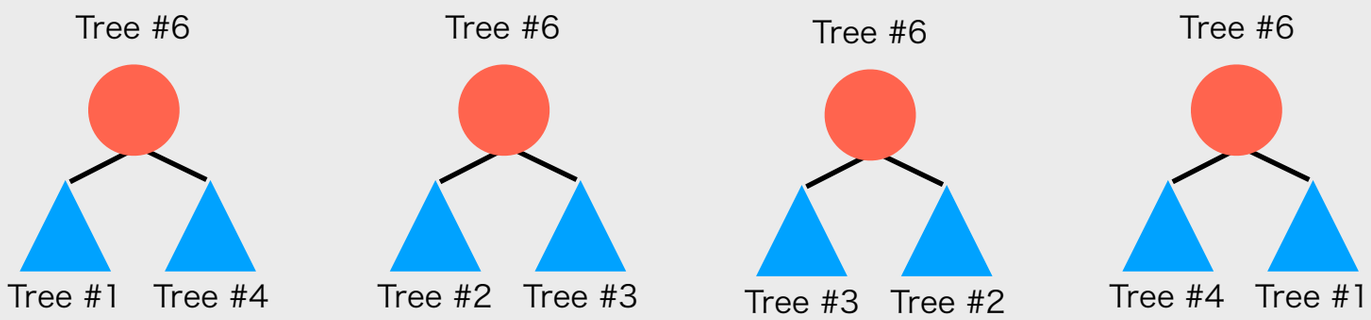
- ノードの型として再帰データ型を用いることが可能
- ノード定義式中は定数サイズパラメータの出現のみを許す
- **cast**式は実行時の正確なサイズを元にサイズの変換(ダウンキャスト)を行う．成功時には変換先の型を持つ変数に束縛される.
- 上の例では**cast**式と**@last**オペレータを組み合わせてデータ構造に対する追加処理を記述している

使用メモリの決定(モジュール)

- ノードの定義式は定数サイズパラメータのみ出現を許す
- それぞれのノード定義式を起点とし更新処理に必要なメモリ量を決定する
- モジュール全体でのメモリ使用量は以下の合計で与えられる
  - それぞれのノードの現在値，直前値を保持するメモリ容量
  - ノードの更新処理に要求される最も大きいメモリ容量
- ノードの更新処理の最後にガベージコレクションを起動し，使用済みのメモリは解放される

使用メモリの決定(ノード)

- 型検査・制約検査後，実行をシミュレーションして使用メモリ量を決定する
  - ノードを起点に関数を呼び出しごとに構文木を展開
  - コンストラクタの呼び出し回数，関数呼び出しのネスト回数などを数える
- 構文木の展開は有限回で停止する
  - サイズ制約の検査により，プログラムの停止は保証されている
- **case**式ではサイズの割り振りを全通り試して要求された最大のリソースを割り当てる
  - 例: **Tree #6**型を持つ値**N(l,r)**については以下の構造を全て探索する



まとめ

- Emfrpに対し再帰データ型，再帰関数を導入した
  - サイズパラメータを持つ型の導入
  - 原始再帰関数の導入
- 構文，型付け規則，操作的意味論を形式的に示した
- 使用メモリ量の決定アルゴリズムを示した

- 今後の課題
  - 型システムの健全性の証明
  - 実装，実験
    - コンパイル時間，実行時オーバーヘッドの計測
    - 近似アルゴリズムの精度の確認
  - メモリ量決定アルゴリズムの計算量削減