SWEST 25 インタラクティブセッション

対話的な組込み用FRP言語インタプリタとFRP言語における省電力コプロセッサの活用

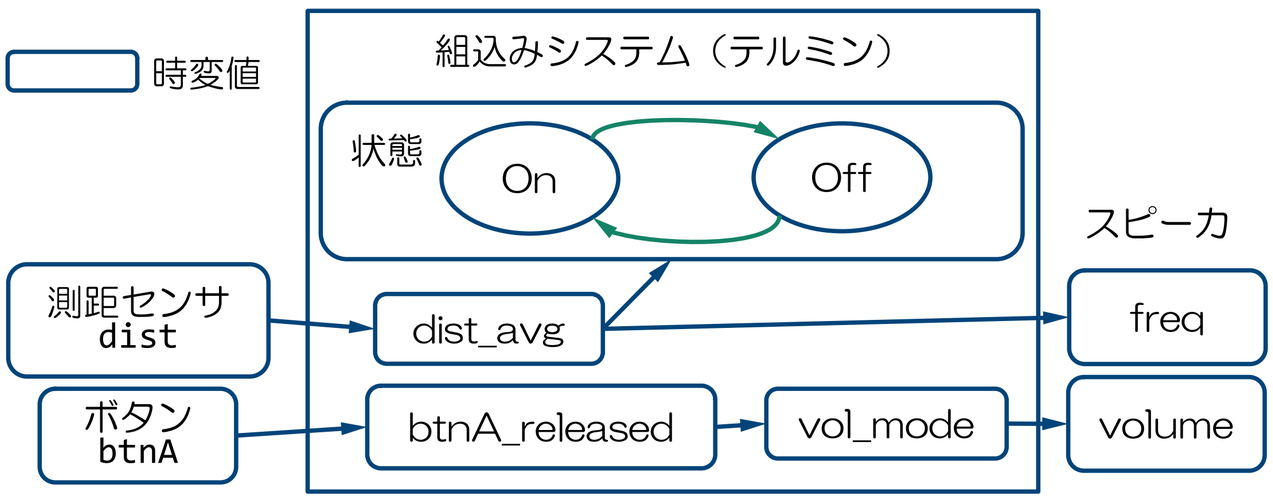
鈴木豪，横山陽彦，森口草介，渡部卓雄（東京工業大学）

組込み用関数リアクティブ（FRP）言語

関数リアクティブプログラミング（FRP）はリアクティブシステムを記述するためのプログラミングパラダイムである．FRPはポーリングやコールバックを用いる必要がなく，可読性を向上させる．**時変値**（**シグナル**）は時間によって変化する値を抽象化したものである．いくつかの組込みシステムは，入力の時変値（センサの値）に反応して出力の時変値（LEDやスピーカなど）を変えるリアクティブシステムとして見ることができる．

我々が設計と実装をしたFRP言語であるEmfrpやXStormでは，時変値は**node**と言う．Emfrp-REPLを除く我々の言語では，再帰的なデータ型や反復処理に制限がある．そのため，コンパイル時にメモリ消費量や実行時間の上限を決定できる．フィードバックループは，**@last**によって直前の時変値の更新（**イテレーション**）による値（**直前値**）を参照して実現する．

右のXStormプログラムは電子楽器であるテルミンのプログラム例である．測距センサによって周波数が変わり，手がある距離を超えて離れると音が止む．また，ボタンによって音量を切り換えることができる．



XStormのプログラムのダイアグラム

本発表では，Emfrpライクな言語のインタプリタであるEmfrp-REPLと，状態によって実行するプロセッサを切り換えることができるXStorm-Coproについて発表する．

```
func next_vol_mode(v) = (v % 5) + 1
switchmodule Theremin {
  in dist : Int, btn(False) : Bool
  out freq : Int, volume : Int
  shared btn_released : Bool, vol_mode(1) : Int
  init On

shared node btn_released = not(btn) && btn@last
shared node vol_mode = if btn_released then
  next_vol_mode(vol_mode@last) else vol_mode@last
state On {
  node dist_avg(0) = (dist*6 + dist_avg@last*4)/10
  out node freq = dist_avg
  out node volume = vol_mode * 20
  switch: if dist_avg >= 1500 then Off() else Retain
}
state Off {
  out node freq = 0
  out node volume = 0
  switch: if dist < 1500 then On() else Retain
}}
```

直前値

XStorm Compiler

```
void input(int * dist,
           int * btn) {
  // ユーザが記述
}
void output(int * freq,
            int * volume) {
  // ユーザが記述
}
EXPORT INT usermain(void)
{ // メイン関数
  activate();
}
```

```
// XStormが出力する
   プログラムのイメージ
int activate(void) {
 while(1) {
  input(&dist, &btn);
  switch(state) {
  case On:
   dist_avg = (dist*6 +
        dist_avg*4) / 10;
      …
  }
  output(&freq, &volume);
}}
```

対話的な組込み用FRP言語インタプリタ

Emfrp-REPLは，EmfrpライクなFRP言語のインタプリタである．動的型付けであることがEmfrpと大きく異なる．MicroPythonのように，REPL（Read-Eval-Print Loop）による対話的な開発ができ，ラピッドプロトタイピングを支援する．

<u>オブジェクトの表現</u>
少ない要素数のタプルなどのオブジェクト表現を工夫し，メモリ消費量を削減した．

<u>実時間ごみ集め</u>
周期的な処理を行うFRP処理系において，実時間性は極めて大事である（次のイテレーションの時間に間に合わないなど）．ゆえに，実時間ごみ集めであるスナップショットごみ集めを用いた．

<u>プログラムの解釈</u>
バイトコードに変換せず，構文木をそのまま解釈する．（ワンパス）

<u>不要な変数のシンボルテーブルの生成の回避</u>
変数の名前解決のためのシンボルテーブルの生成はコストがかかる処理である．例えばMicroPythonでは，バイトコードへの変換時にシンボルテーブルの生成の回避をする最適化をしている．FRP言語インタプリタであるEmfrp-REPLはノードの評価時にそのようなシンボルテーブルが不要であることに着目し，ワンパスで不要なシンボルテーブルの生成の回避をする．

```
1 > node dist_avg init[dist] = (dist * 2 +
    dist_avg@last * 8) / 10
    > OK, NIL
2 > node freq = dist_avg
    > OK, NIL
3 > (dist_avg, dist)
    > OK, (30, 80)
4 > node dist_avg = (dist * 6 + dist_avg@last * 4) /
    10
    > OK, NIL
5 > node is_output = dist_avg > 1500
    > OK, NIL
6 > node volume = if is_output then 100 else 0
    > OK, NIL
7 > node dist_avg = if is_output then (dist * 6 +
    dist_avg@last * 4) / 10 else dist_avg@last
    > ERROR, Cyclic Refrence

8 > node dist_avg = if is_output@last then (dist * 6
    + dist_avg@last * 4) / 10 else dist_avg@last
    > OK, NIL
```

反応が遅い気がするので値を確認しよう

ノードを再定義

循環定義エラー

FRP言語における省電力コプロセッサの活用

XStorm-Coproは，XStormを拡張し，省電力コプロセッサを活用できるようにした．ESP32などは，機能（ペリフェラルの利用など）と性能が制限された省電力コプロセッサを持つ．メインプロセッサを寝かせ，省電力プロセッサで入出力を扱うことで，反応性を維持したまま省電力化を可能にする．

<u>提案</u> 状態ごとに実行するプロセッサを指定できるようにする．

<u>利点</u>
- FRPにおいて，ULPコプロセッサを活用できる．すなわち，省電力化に貢献する．
- メインプロセッサとコプロセッサ間のデータ転送処理を自動的にコンパイラが生成し，開発者の負担を下げる．
- プロセッサの状態管理を自動的に行うので，それに起因するバグを軽減することができる．

<u>ターゲット</u> RISC-V Ultra Low Power Coprocessor（ULPコプロセッサ）を搭載したESP32-S3

ESP32-S3のULPコプロセッサは，AD変換やI²Cに対応しているため，メインプロセッサでなくとも測距センサの値を取得できる．よって，テルミンの例では，状態がOffの場合，ULPコプロセッサで実行することで，省電力化が期待できる．（ULPコプロセッサでPWM出力はできないため，状態Onはメインプロセッサで実行する必要がある．）

```
switchmodule Theremin {
  …

state On on main {
  node … = …
}

state Off on ulp {
  node … = …
}
}
```

メインプロセッサのためのCのプログラム

メインプロセッサとULPコプロセッサ間のデータ転送処理

ULPコプロセッサのためのCのプログラム