数理解析研究所講究録
第2005巻 2016年 84-96

偏微分方程式の陽解法プログラムの自動生成と自動チューニング言語Paraisoについて

理化学研究所・計算科学研究機構 村主崇行

Takayuki Muranushi
RIKEN Advanced Institute for Computational Science
7-1-26, Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, 650-0047, Japan
takayuki.muranushi@riken.jp

この文章の各章はLiterate Haskellとして解釈可能で、日本語による解説であると同時に実行可能なプログラムになっています。人間とコンピュータへともに語りかける文章であり、知識が構築される家庭を明文化することでOpen Knowledgeへの貢献を試みるものです。これらの文章は`http://en.pk.paraiso-lang.org/`からも閲覧でき、ソースコードは`https://github.com/nushio3/pkwiki`から取得できます。

1 はじめに

コンピュータシミュレーションは自然科学にとって欠かすことができません。先人の業績、ライブラリの積み重ね、ソフトウェア開発に関する知見の蓄積、3年で4倍、という指数関数的ペースで成長するコンピュータの演算性能の恩恵のもと、過去には不可能であった計算を、可能にしてみせる歴史でした。とくに、歴史の綾により画像処理装置から進化したGPGPU(General Purpose Graphic Processing Unit)の登場以来、パソコンにおいても何万スレッドを用いた並列計算は当たり前にできるようになり、世界のトップレベルのスーパーコンピューターでは、何億という演算器を同時に操るプログラミングが行われるようになってきています。

その反面、いまや高度に発達した計算機械の上でのプログラミングの苦労は大変なものになっています。より高度な、複雑な物理が入ったシミュレーションが行われるようになったことと、計算機が複雑度、並列度が増していったこととの相乗効果。それに加えて、新しいハードウェアや計算パラダイムが登場するたびに、プログラムの全面的書き直しを強いられることが、プログラミング作業を膨大にしています。

もっと賢いやり方はないのか？ 人間がプログラムを書き出すのに使用している知識を体系化してコンピュータに与えれば、この膨れ上がったシミュレーションプログラムの生成を、コンピュータに肩代わりさせられないか。解きたい問題の定義を与えれば、プログラムが出てくるようにし、人間は本質的、創造的な問題の探求に注力できないか。万能のプログラム生成は無理でも、シミュレーション分野を限った言語(DSL, Domain Specific Language)なら、挑みうる問題なのではないか。

多くの人がそう考え、試みてきました。実際、Paraiso と競合する Stencil 計算の分野でも、 LibGeoDecomp [7] Physis [4] Halide [6] などの試みがなされてきました。

そんな中、筆者が多くのアドバイスをいただきながら独力で実装してきた Paraiso は、マンパワーが少ないことから、対応分野をうんと狭くする代わりに、端から端まで、抽象と機械化の間をなるべく長く結ぶことを目指しました。数学的記法による偏微分方程式の簡潔な記述から、プログラムを自動生成し、そして遺伝的アルゴリズムと焼き鈍しに基づくプログラム自動最適化により、数倍から 10 数倍の性能向上を自動的にもたらしうることを実証してみせました。荒削りで、人が通れるようになるのはまだまだ先ですがシミュレーション科学者の楽園 (Paraiso) へと続く隧道にツルハシを入れてみせたのです。

2 はじめての直胞機械プログラミング

Paraiso は、一様メッシュ上での偏微分方程式の陽解法という分野に特化し、その分野の計算を、直胞機械 (OM, Orthotope Machine の訳語) と呼ばれる、n 次元配列を操作する仮想並列機械のプログラムとして表現します。直胞機械は、配列変数とその操作を表現する仮想機械であり、解きたい問題の内包する並列性をなるべく生かすように設計されています。たとえば、配列変数の要素を格納する順序や、アクセスする順序は自由。命令どうしの依存関係もはじめから明確で、依存性のない命令を実行する順序も、同時に実行することも自由です。Paraiso は、この直胞機械のプログラムを記述し、それを既存の言語のプログラムに翻訳し、さらに自動チューニングを行うための様々なメカニズムを備えたフレームワークです。より詳しい用語の解説などは Paraiso の論文 [5] を見てください。

しかし、Paraiso にコードを生成させるためにユーザーがすべきことを、具体的に一言でいうなら、 「`OM` 型の値を作り、コードジェネレータに渡すこと」となります。

2.1 ハローワールドの中身

本節で掲載するサンプル `examples/HelloWorld/Generator.hs` には、Paraiso にコードを生成させるための全てが記されています。順にみていきましょう。

最初の十数行は`#include <stdio.h>`みたいなもんで、必要なモジュールを導入する文が並んでいます。この部分は情報密度が少ないので、割愛させていただきます。オンライン版を見てくださいね。では `main` から見ていきましょう。

```
> -- the main program
> main :: IO ()
> main = do
>   _ <- system "mkdir -p output"
>   T.writeFile "output/OM.txt" $ prettyPrintA1 $ myOM
>   _ <- generateIO mySetup myOM
>   return ()
```

一行目は`"output"`というフォルダを作っています。二行目では、`myOM` という名の直胞機械のデータフロー・グラフを出力しています。これは参考のために出力しているのであって、コード生成に必須の過程ではありません。三行目では、`myOM`

の定義と mySetup というコード生成時の設定を渡して C++のコードを生成しています。

直胞機械を定義する時には使わないが、コード生成時にはじめて必要になる情報は Language.Paraiso.Generator.Native モジュールにある Setup 型でまとめて指定します。ここでは 具体的な計算のサイズや、ライブラリを生成するフォルダを指定しています。コード生成対象の言語はデフォルトでは C++です。

```
> -- the code generation setup
> mySetup :: Native.Setup Vec2 Int
> mySetup =
>   (Native.defaultSetup $ Vec :~ 10 :~ 20)
>   { Native.directory = "./dist/"
>   }
```

次はいよいよ myOM の定義です。 直胞機械を定義するには、名前 (tableMaker)、マシン全体の注釈、静的変数のリスト、カーネルのリストの 4 つを makeOM 関数に渡します。

```
> -- the orthotope machine to be generated
> myOM :: OM Vec2 Int Annotation
> myOM = optimize O3 $
>   makeOM (mkName "TableMaker") [] myVars myKernels
```

myOM には、九九の表を保持するための table と、その合計を計算するための total という 2 つの変数があります。2 つの変数の Realm は、それぞれ Array と Scalar です。Array 変数は解きたい問題の解像度とおなじサイズをもつ配列変数で、Scalar 変数は普通に 1 つしかない値です。それぞれの変数は Name も持っています。

```
> -- the variables we use
> table :: Named (StaticValue TArray Int)
> table = "table" `isNameOf` StaticValue TArray  undefined
>
> total :: Named (StaticValue TScalar Int)
> total = "total" `isNameOf` StaticValue TScalar undefined
>
> myVars :: [Named DynValue]
> myVars = [f2d table, f2d total]
```

Paraiso では、 Language.Paraiso.Name モジュールで定義されている Name 型を識別子の名前として使います。これは、ただの文字列と区別するために、Data.Text モジュールの提供するマルチバイト文字列型 Text を newtype したものです。

次にカーネルの定義です。OM のカーネルとは、一度に実行される計算のかたまりです。myOM には create という名前のカーネルがひとつだけあります。

```
> -- the only kernel our OM has
> myKernels :: [Named (Builder Vec2 Int Annotation ())]
> myKernels = ["create" `isNameOf` createBuilder]
```

カーネルを作るには Builder Monad を使います。ここでは、x 軸と y 軸の添字を読み込んでその積を計算するカーネルを作っています。ついでに九九の表の総計も求めてみましょう。

```
> -- the kernel builder monad
> createBuilder :: Builder Vec2 Int Annotation ()
> createBuilder = do
>   x <- bind $ loadIndex (Axis 0)
>   y <- bind $ loadIndex (Axis 1)
>   z <- bind $ x*y
>   store table z
>   store total $ reduce Reduce.Sum z
```

3 はじめてのGPUプログラミング

それでは、いよいよGPUプログラミングをやってみましょう。C++コードの生成器をCUDA用のもの. に書き換えるのは、実際すごく簡単です。

```
$ diff HelloWorld/Generator.hs HelloGPU/Generator.hs
40c40,42
<   { Native.directory = "./dist/"
---
>   { Native.directory = "./dist/" ,
>     Native.language  = Native.CUDA,
>     Native.cudaGridSize = (32,1)
```

このように、ネイティブコード生成の詳細を指定する `Setup` 型の値において、CUDAコードを生成したいということと、CUDAのグリッドサイズを指定するだけです。(以下の指定の場合、すべてのCUDAカーネルは`<<<32,1>>>`のサイズをもって起動されます。)

```
> mySetup :: Native.Setup Vec2 Int
> mySetup =
>   (Native.defaultSetup $ Vec :~ 10 :~ 20)
>   { Native.directory = "./dist/" ,
>     Native.language  = Native.CUDA,
>     Native.cudaGridSize = (32,1)
>   }
```

生成されたコードは東京工業大学のTSUBAME 2.0で、nvcc 4.1(と内臓されたthrust)を用いて、C++版と同じ出力をすることを確かめました。

4 Builderモナド

直胞機械のデータフロー・グラフの型は、`vector`、`gauge`、`anot`という3つの型引数を取るもので、これがBuilderにも常についてまわります。`vector`は配列の次元(例:3次元)、`gauge`は添字の型(例:Int)、`anot`は解析や最適化に使う注釈の型でして、`vector gauge`という組み合わせは配列にアクセスするための添字のベクトル(Intの3次元ベクトル)になります。

```
> type Graph vector gauge anot = Gr (Node vector gauge anot) Edge
```

Paraiso のプログラムは、`Builder` モナドを組み合わせることで記述します。`Builder` モナドは、実のところ作りかけのデータフローグラフを状態にもつ `State` モナドです。それぞれの `Builder` モナドは、データフローグラフの頂点をいくつか引数に取って、新たな頂点を返します。

```
> -- | value type, with its realm and content type discriminated in
> --   type level data
>   Value rea con =
>   -- | data obtained from the data-flow graph.
>   -- 'realm' carries a type-level realm information,
>   -- 'content' carries only type information and its ingredient is
>   -- insignificant and can be 'undefined'.
>   FromNode {realm :: rea, content :: con, node :: G.Node} |
>   -- | data obtained as an immediate value.
>   -- 'realm' carries a type-level realm information,
>   -- 'content' is the immediate value to be stored.
>   FromImm {realm :: rea, content :: con} deriving (Eq, Show)
```

この頂点は `Value` 型であり、各頂点にたいして領域情報 `rea` と、中身の型の情報 `con` を運んでいます。`rea` の位置に入るのは Language.Paraiso.OM.Realm にある `TArray` か `TScalar` のいづれかであり、それぞれ配列変数およびスカラー変数を表します。`con` の位置に入るのは `Int` とか `Double` とかいった要素型ですが、多くの場合は型情報のみがグラフの構築に使われ値は単に無視されます。この多相な [`Value` 型] を接点にグラフを組み立てることで、できあがったグラフの型安全性が (かなり) 保証されます。

Paraiso プログラムを組み立てるモナド材料は Language.Paraiso.OM.Builder モジュールに揃っています。モジュールのページに行って、“Synopsis” タブを押してください。

4.1 bind が必要なわけ

これまでもいくつか Paraiso のプログラムはでてきましたが、やたらと各行ごとに `bind` があるなあと思われたかもしれません。

```
> bind :: (Monad m, Functor m) => m a -> m (m a)
> bind = fmap return
```

こんなふうに。

```
>   x <- bind $ loadIndex (Axis 0)
>   y <- bind $ loadIndex (Axis 1)
>   z <- bind $ x*y
>   w <- bind $ x+y
```

どうしてこうなるのでしょう？ このプログラムをもっと素直に、以下のように書けないものでしょうか。

```
>   x <- loadIndex (Axis 0)
>   y <- loadIndex (Axis 1)
>   z <- return x * return y
>   w <- return x + return y
```

ところが、右辺の式 `loadIndex (Axis 0)` などは `Builder` モナドから成り立っている一方で、左辺の `x,y,z,w` などはデータフローグラフの頂点ですから、`Value` 型の値です。このため、いったん左辺で `x` などの変数名を束縛したら、その後右辺で使うたびにモナドに変換する必要があります。(このとき使うのはむろん、最小の文脈を付与する `return` です！) ところが `bind = fmap return` により束縛時点で一度 `return` を施せば、以降、右辺ではそのまま使えます。こちらの方がずっと便利でしょう？

というわけで、Paraiso のサンプルプログラムに出てくる `x,y` といった何気ない変数名はすべてモナディックな値であることに注意して下さい。

さらに、この `bind` には Sharing discovery（重複する式を発見し同一化する）の効果もあります。実のところモナドを介して Sharing をするというのはずっと昔に試みられて捨てられてたりします。しかし、Haskell のモナドを扱う能力がずいぶん進歩した今となっては、ここでモナドを使うのはごく自然なことです。むろん、`<-bind $`というパターンが並ぶのは醜いので、将来的にはなんらかの QuasiQuote でも作りたいものです。あと完全な Sharing discovery を実現するにはノード追加時のチェックが必要です。

4.2 直胞機械の命令セット

直胞機械の (実際コンパクトな) 命令セットは `Language.Paraiso.OM.Graph` モジュールの `Inst` 型として定義されており、`Language.Paraiso.OM.Builder` モジュールには、これにほぼ一対一対応する形で材料が用意してあります。以下に、それぞれのコンビネータを表形式で整理しておきます。`B` を一般のモナド記号 `m` だと (実際の定義は `type B ret = forall v g a. Builder v g a ret`) 思うと感覚がつかめるのではないでしょうか。

これが命令セットで、

```
> data Inst vector gauge
>   = Load StaticIdx
>   | Store StaticIdx
>   | Reduce R.Operator
>   | Broadcast
>   | LoadIndex (Axis vector)
>   | LoadSize (Axis vector)
>   | Shift (vector gauge)
>   | Imm Dynamic
>   | Arith A.Operator
```

これが対応するモナド・コンビネータです。

```
> --           options                    input nodes            output nodes
> load      :: Named (StaticValue r c)                       -> B (Value r c)
> store     :: Named (StaticValue r c) -> B (Value r c)      -> B ()
> reduce    :: Reduce.Operator         -> B (Value TArray c) -> B (Value TScalar c)
> broadcast ::                            B (Value TScalar c) -> B (Value TArray c)
> loadIndex :: Axis v                                        -> B (Value TArray g)
> loadSize  :: Axis v                                        -> B (Value TScalar g)
> shift     :: v g                     -> B (Value TArray c) -> B (Value TArray c)
> imm       :: c                                             -> B (Value r c)
```

各命令の機能を端的に言うと、

- `load` は名前つき静的変数から読み込む。
- `store` は名前つき静的変数へ書き込む。
- `reduce` は配列をスカラーに畳み込む。
- `broadcast` はスカラーを配列にする。
- `loadIndex` は配列の添字を取得。
- `loadSize` は配列のサイズを取得。
- `shift` はベクトル `v g` を使って配列をずらす。
- `imm` は即値を読み込む。
- `Arith` はさまざまな算術演算を施す。

となります。各命令の詳細については、これから例で見ていきます。

4.3 演算子たち

そういえば、`Arith` 命令に対応するコンビネータが見当たりませんが、どうなってるのでしょうか。Paraiso では、numeric-prelude という Haskell の代数拡張ライブラリを使い、Buider モナドをさまざまな代数構造のインスタンスにすることで、普通の数式を書くのと同じ感覚で Buider モナドの数式を組み立てられるようになっています。

```
> (+) :: B (Value r c) -> B (Value r c) -> B (Value r c)
> sin :: B (Value r c) -> B (Value r c)
```

ただし、Haskell の論理値 (Bool) を扱う演算子に関しては、`(==) :: Eq a => a -> a -> Bool` などの固定された型を持っているため、Bool の Builder を返させることができません。そのため、モジュール Language.Paraiso.OM.Builder.Boolean や Language.Paraiso.Prelude. で定義されている関数で代用してください。また `if` も同じ理由で Builder を組み立てる用途には使えませんので、代わりに `select` を使ってください。

```
> eq :: B (Value r c) -> B (Value r c) -> B (Value r Bool)
> ne :: B (Value r c) -> B (Value r c) -> B (Value r Bool)
> select :: B (Value r Bool) -> B (Value r c) -> B (Value r c) -> B (Value r c)
```

最後に、型変換のための関数 `cast` と `castTo` があります。具体的にどういう型変換コードが生成されるかは対象言語しだいです。`cast` は単に型変換する関数で変換先の型の指定は型推論に任せます。これに対し `castTo` は引数 `c2` で変換先の型を指定できます。`c2` の値は使われません。

```
> cast   :: B (Value r c1) -> B (Value r c2)
> castTo :: c2 -> B (Value r c1) -> B (Value r c2)
```

5 Paraiso の境界条件

直胞機械の `shift` 命令は配列変数全体を平行移動する命令です。実際のコードが扱う配列変数は常に有限サイズですから、平行移動によりはみ出す部分の処理を

境界条件により指定してやる必要があります。Paraiso は、データフローグラフの中の `shift` 命令を解析して、境界条件を実現するのに必要な処理 (袖領域など) を追加してくれます。

Paraiso では、今のところ周期境界と自由境界という二種類の境界条件をサポートしています。コード生成時、次元ごとにいづれかの境界条件を選ぶことができます。この二種類とほかの命令を組み合わて作れる範囲で、より複雑な境界条件も扱えます。境界条件は、`Setup` データ型の `boundary` レコードに対して `Condition` 型を要素にもつベクトルを与えることで指定します。

二種類の境界条件のうち、周期境界は、単に右端からはみ出したアクセスは左端のものを見るだけです。いわゆるドーナツ惑星の地図です。

これに対し、自由境界の場合は少々複雑です。まず、袖領域は、どの Paraiso カーネルを 1 ステップ実行しても、Setup で指定した計算したい領域、添字が 0 以上 `localSize` 未満の領域が埋めるのに必要十分なサイズをもって決まります。次に、各カーネルの計算する範囲は、袖領域をふくめた最大の領域から開始し、なるべく大きな領域を埋めるように決まります。その領域の外側での値は未定義です。また、`Reduce` 演算はつねに袖領域をふくむ全体を畳み込みます。もし袖領域を除外したい場合は手動でマスクする必要があります。

このサンプルプログラムで境界条件の挙動を確かめてみましょう。このプログラムには 3 つのカーネルがあります。

```
> myKernels :: [Named (Builder Vec1 Int Annotation ())]
> myKernels =
>   ["init" `isNameOf` do
>       store table $ loadIndex (Axis 0)
>
>   ,"increment" `isNameOf` do
>       store table $ 1 + load table
>
>   ,"calculate" `isNameOf` do
>       center <- bind $ load table
>       right  <- bind $ shift (Vec :~ (-1)) center
>       left   <- bind $ shift (Vec :~ ( 1)) center
>       ret    <- bind $ 10000 * left + 100 * center + right
>       store table ret
>       store total $ reduce Reduce.Sum ret
>   ]
```

それぞれ、「配列の内容を配列添字で初期化する」「配列の内容に 1 を加える」「ひとつ右とひとつ左の内容を読んで値を計算する」カーネルです。

```
maker.init();
dump();
maker.increment();
dump();
maker.calculate();
dump();
cout << "total: " << maker.total() << endl;
cout << endl;
```

`make` を実行すると、境界条件だけが異なる 2 種類のプログラムが生成されます。以下のように、周期境界条件のもとでは境界を飛び越えて反対側の値にアクセスできます。

```
$ ./main-cyclic.out
index:      0      1      2      3      4      5      6      7
value:      0      1      2      3      4      5      6      7
index:      0      1      2      3      4      5      6      7
value:      1      2      3      4      5      6      7      8
index:      0      1      2      3      4      5      6      7
value:  80102  10203  20304  30405  40506  50607  60708  70801
total: 363636
```

一方、自由境界のもとでは袖領域が追加され、0 未満、あるいは `localSize` 以上の添字の範囲にもアクセスできることがわかります。

```
$ ./main-open.out
index:     -1      0      1      2      3      4      5      6      7      8
value:     -1      0      1      2      3      4      5      6      7      8
index:     -1      0      1      2      3      4      5      6      7      8
value:      0      1      2      3      4      5      6      7      8      9
index:     -1      0      1      2      3      4      5      6      7      8
value:      0    102  10203  20304  30405  40506  50607  60708  70809      0
total: 283644
```

6　人生っていいもんだ

Paraiso を実用的な問題で試してみましょう。そうですね、生命現象のシミュレーション [2] などが実用的で良いのではないでしょうか。さっそくサンプルプログラムを見ていきましょう。まずインポートは省いて、`main` 関数です。

```
> -- the main program
> main :: IO ()
> main = do
>   _ <- generateIO mySetup myOM
>   return ()
```

80 マス× 48 マスの大きさで、周期境界条件が課された計算領域を用意します。OM の名前はずばり “Life” にしましょう。

```
> -- the code generation setup
> mySetup :: Native.Setup Vec2 Int
> mySetup =
>   (Native.defaultSetup $ Vec :~ 80 :~ 48)
>   { Native.directory = "./dist/" ,
>     Native.boundary = compose $ const Boundary.Cyclic
>   }
>
> -- the orthotope machine to be generated
> myOM :: OM Vec2 Int Annotation
> myOM = optimize O3 $
>   makeOM (mkName "Life") [] myVars myKernels
```

セルオートマトンの状態を表す配列変数 `cell`、人口を数えるためのスカラー変数 `population`、および経過時間を数えるためのスカラー変数 `generation` という三つの変数を用意します。

```
> -- the variables we use
> cell :: Named (StaticValue TArray Int)
> cell = "cell" `isNameOf` StaticValue TArray  undefined
>
> population :: Named (StaticValue TScalar Int)
> population = "population" `isNameOf` StaticValue TScalar undefined
>
> generation :: Named (StaticValue TScalar Int)
> generation = "generation" `isNameOf` StaticValue TScalar undefined
>
> myVars :: [Named DynValue]
> myVars = [f2d cell, f2d population, f2d generation]
```

カーネルは 2 つ作りましょう。1 つは初期化のためのカーネルで、もうひとつはシミュレーションを 1 ステップ進めるためのカーネルです。

```
> -- our kernel
> myKernels :: [Named (Builder Vec2 Int Annotation ())]
> myKernels = ["init" `isNameOf` initBuilder,
>              "proceed" `isNameOf` proceedBuilder]
```

初期化は、ただのゼロ初期化です。

```
> initBuilder :: Builder Vec2 Int Annotation ()
> initBuilder = do
>   -- store the initial states.
>   store cell 0
>   store population 0
>   store generation 0
```

さて、ライフゲームのルールをどのように書けばいいのでしょうか。まずは、セルの隣接関係を定義するところから始めましょう。Conway のライフゲームでは、あるセルの縦、横、斜め 8 セルを隣接していると定義します。

```
> adjVecs :: [Vec2 Int]
> adjVecs = zipWith (λ x y -> Vec :~ x :~ y)
>           [-1, 0, 1,-1, 1,-1, 0, 1]
>           [-1,-1,-1, 0, 0, 1, 1, 1]
```

“cell” という名の配列変数から、ステップ開始前のセルの状態を読み取ります。

```
> proceedBuilder :: Builder Vec2 Int Annotation ()
> proceedBuilder = do
>   oldCell <- bind $ load cell
```

それから、“generation” という名のスカラー変数も必要なので読み込んでおきましょう。

```
>   gen  <- bind $ load generation
```

`shiftedCell <- bind $ shift v oldCell` とすると、もとのセル配列をベクトル `v` だけずらした配列が手に入りますよね。これを **`adjVecs`** に入っているすべての `v` について繰り返すことで、すべての隣接セルの情報を集められます。

```
>   neighbours <- forM adjVecs $
>     λ v -> bind $ shift v oldCell
```

生きている隣接セルの数を数えましょう。

```
>   num <- bind $ sum neighbours
```

「生きているセルは、隣人が 2 人か 3 人だったら生き延びる。死んだセルは、隣人が 3 人だったら生き返る」という Conway のライフゲームのルールを適用します。

```
>   isAlive <- bind $
>               (oldCell `eq` 0) && (num `eq` 3) ||
>               (oldCell `eq` 1) && (num `ge` 2) && (num `le` 3)
```

ルールの判定にもとづき、セルの状態を更新します。`select isAlive 1 0` という式は、`isAlive` の真偽値に応じて 1 か 0 かの値をとります。C 言語風に訳すと `isAlive ? 1 : 0` です。

```
>   newCell <- bind $ select isAlive 1 0
```

最後に、次のステップの状態を保存します。

```
>   store population $ reduce Reduce.Sum newCell
>   store generation $ gen + 1
>   store cell $ newCell
```

いかがでしたでしょうか。背景のモナディックな枠組みをほとんど意識せずに式が書けている所もあれば、舞台装置な処理を意識させてしまっている箇所もあると思います。かつてはモナドを使った DSL とかめどい [1] という意見もありましたが、Haskell のモナドを扱う能力はずいぶん進歩しています。私もすごい Haskell 本 [3] の訳を手がけたおかげで、だんだん書き方が分かってきました。いろんなライブラリがモナドの言葉でかかれているので、皆さんがそんなライブラリを作ったり使ったりするとき、すごい Haskell 本もお役に立てれば幸いです。

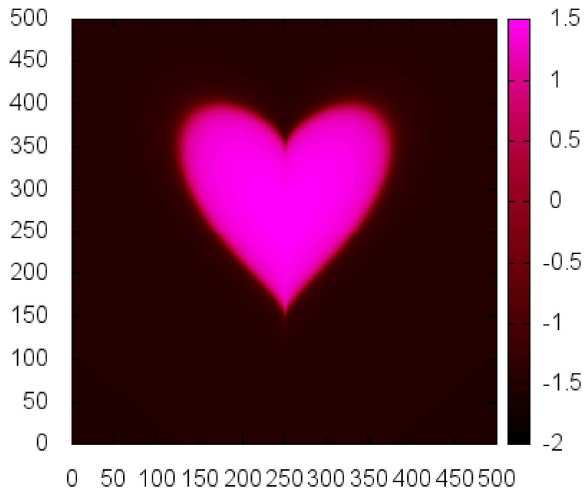
7 Paraiso でシミュレーションを始める

シミュレーションを始めるには、まず初期条件を設定できる必要があります。初期条件の場を計算する仕事は、Paraiso 式づくりのとっかかりとしても優れています。やってみましょう。サンプルを見てください。

```
> table :: Named (StaticValue TArray Double)
> table = "table" `isNameOf` StaticValue TArray undefined
>
> createBuilder :: Builder Vec2 Int Annotation ()
> createBuilder = do
>   x01 <- bind $ (cast $ loadIndex (Axis 0))
>               / (cast $ broadcast $ loadSize (Axis 0))
>   y01 <- bind $ (cast $ loadIndex (Axis 1))
>               / (cast $ broadcast $ loadSize (Axis 1))
>   x <- bind $ 4 * (x01-0.5)
>   y <- bind $ 5 * (y01-0.5)
>   z <- bind $ atan((1- x^2 - (y-(x^2)**(1/3))^2)*10)
>   store table z
```

`Generator.hs` では、少々煩雑ですが、`cast` と `broadcast` を使って配列変数の添字とサイズから、x 座標と y 座標を 0 から 1 までの範囲の実数として求めています。 `cast` や `broadcast` の引数や返値は、うまいこと型推論してもらえてるみたいですね。実際、配列変数の添字とサイズは `createBuilder` の型注釈からして `Int` 型なのに対し、`table` は `Double` 型で、そこから明示的な `cast` を挟まずに繋がっている `x01`, `y01`, `x`, `y`,`z` などの型もみな `Double`(の Builder) になります。経験のある読者は、`1/3` が一見整数どうしの除算として `0` に切り捨てられることなく、ちゃんと `Double` 同士の除算として扱われていることにも気づいたかもしれません。

それでは、これが Paraiso と筆者から、読者へ、そして数理研研究集会にて発表の機会を与えてくださった皆様、聞いてくださった皆様への気持ちです！



$$f(x,y) = \mathrm{atan}\left((1 - x^2 - 10(y - x^{2/3})^2)\right) \tag{1}$$

References

[1] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Advances in Computing Science—ASIAN' 99*, pages 62–73. Springer, 1999.

[2] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[3] M. Lipovača, H. Tanaka, and T. Muranushi. すごい *Haskell* たのしく学ぼう! 株式会社 オーム社, 2012.

[4] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[5] T. Muranushi. Paraiso: an automated tuning framework for explicit solvers of partial differential equations. *Computational Science & Discovery*, 5(1):015003, 2012.

[6] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

[7] A. Schäfer and D. Fey. Libgeodecomp: A grid-enabled library for geometric decomposition codes. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 285–294, Berlin, Heidelberg, 2008. Springer-Verlag.