チュートリアル

プログラミング言語 Haskell とその処理系

尾上能之

1 はじめに

プログラミング言語をパラダイムとして分類するとき，手続き型，関数型，論理型，対象指向型などがよく知られている．これらのうちで，関数プログラミングの考え方は古くから存在するが，十分に普及しているとは言えない．その理由として，従来は標準的な言語や処理系が定まっていないことから広く認知されるまでに至らなかったことに加え，処理系の多くは実行速度も遅くほとんど実用に耐えなかったことに由来する．ところが近年では，標準言語として Haskell の仕様が定められ，処理系も十分に高速なものができつつあるのが現状である．

そこで本稿では，関数型言語 Haskell の特徴を解説し，現在利用可能な処理系の中で広く用いられている Hugs インタプリタと GHC コンパイラについて，その使い方などを紹介する．

2 関数型言語 Haskell

2.1 特徴

Haskell は，1987 年に開かれた国際会議 FPCA (Functional Programming Languages and Computer Architechture) において，類似した関数型言語が複数存在していることが関数プログラミングの普及を妨げていると考えられたことから，標準言語となるべくして提案された．その名前は論理学者の Haskell B. Curry に由来している．最初に提案された仕様は 1.0 版であったが，その後改良を重ねるにつれ，1.1, 1.2, 1.3, 1.4 となり，現在は 1999 年に提案された Haskell 98 [16] が最新の仕様[†1]となっている．言語仕様や，Haskell に関する書籍，処理系の紹介，Haskell で記述されている応用ソフトなど，Haskell に関するあらゆる情報は，ホームページ (http://haskell.org) によくまとめられているので，ここを一次情報源とするとよいであろう．

Haskell は現代の多くの関数型言語にみられる次のような特徴をすべて備えている [7].

- 高階関数 (high-order function)
- 遅延評価 (lazy evaluation) / 非正格性 (non-strictness)
- データ抽象化と強い型付け
- パターン照合 (pattern matching)

さらに Haskell 独自の特徴としては，以下のものが挙げられる．

1. 多様型 (polymorphism) / 多義性 (overloading) 関数やデータ構造が，引数や返り値の型として複数の型を許容する．多様型をもつ関数は，その引数がどのような型をもつのかということには関係なく，型とは無関係に同じように働く．一方，多義性をもつ関数は，型クラスの概念を用いる

Programming Language Haskell and Systems

Yoshiyuki ONOUE, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, University of Tokyo

コンピュータソフトウェア, Vol.18, No.6(2001), pp.59–66.

[チュートリアル] 2001 年 9 月 14 日受付.

†1 2001 年中に修正版がリリースされる予定

ことによって，型が異なるごとに違った振舞いをする．

2. 関数的入出力システム
   通常，入出力の概念は状態や副作用と密接に関係するものだが，Haskell では Monad の概念を用いることによって，プログラム内における参照透明性を維持したまま入出力を実現している．
3. リスト/配列の内包表記 (list/array comprehension)
   要素をすべて羅列することなく，その性質を用いることによって，リストや配列を簡潔に記述することが可能となる．例えば，1 から 100 までの奇数の 2 乗の集合 $\{x^2|x \in \{1,\ldots,100\}, x \text{ is odd}\}$ は，リストの内包表記を用いると `[ x*x | x<-[1..100], odd x]` のように直接的に記述することが可能である．
4. モジュール機能
   関数やデータ型の集まりをモジュールとして構成的に分離し，相互に参照可能な要素を制御することによって，構造的なプログラミングを可能にしている．

関数型言語は上記のような特徴，すなわち高いレベルでの抽象化によって構造的プログラミングを可能にし，代入などによる副作用のない純粋な式の概念を用いていることから，教育の現場において最初に教えるプログラミング言語としても採用されることが多くなってきている．例えば先述した Haskell ホームページには，大学の講義で，プログラミングの教育をほとんど受けたことがない学生に対して，最初または二番目に教える言語として Haskell を用いている所が 21 コース挙げられている．このように，まだ絶対数は少ないものの，プログラミングの概念を学ぶ言語として，Haskell の有用性が徐々に浸透してきている．

2.2 プログラムの例

Haskell の言語仕様は文献 [16] に定められているが，本稿で詳しく解説することは不可能なので，実例を交えて Haskell のエッセンスを紹介するに留める．Haskell の入門書としては文献 [2] [9] [19] などが有名であるので，興味をもたれた方はこちらも参照していただきたい．

ここでは例として，最小の Ramanujan 数を求めるプログラム [17] を考えよう．Ramanujan 数とは 2 つの自然数の 3 乗の和で 2 通りに表わされる数のことで，

$$x = a^3 + b^3 = c^3 + d^3 \qquad (1)$$

となる自然数で $a \neq c, a \neq d$ となる最小のものを求めたい．文献 [17] ではプログラムが Miranda [20] で記述されているが図 1 では Haskell で書き直してある．

ここではモジュールの概念を説明するため，プログラムを 3 つのモジュールに分解し，それぞれファイル (`Main.hs`, `Rama.hs`, `Sort.hs`) に保存することにした．このように通常 Haskell で書かれたプログラムには，拡張子 `.hs` を用いる．このプログラムを動作させると，`Main` モジュールに含まれる識別子 `main` が評価され，その値が結果として返されることになる．

コメントは，`--` から行末まで (l.10) と，`{-`,`-}` で囲まれた範囲 (ll.27–30) の 2 通りの記法が可能である．

モジュール間では，関数，データ構成子や型などのインポート/エクスポートを制御することによって，構造的プログラミングが可能になっている．この例では l.6 や l.19 において，それぞれ関数 `ramanujan`, `fsort` を外部にエクスポートしている．また l.1 のように括弧による指定がない場合はモジュール内で定義されたすべてのトップレベル関数がエクスポートされる．一方インポートするには l.3 や l.8 のように記述する．l.3 では，モジュール `Rama` でエクスポートされているすべての関数をインポートしている．

条件分岐は，ll.33–35 のように表わす．すなわち，式 `r x <= r y` の値が真になるときは l.34，それ以外の場合は l.35 の右辺式の値が返されることになる．また ll.33 では，第 2,3 引数についてパターン照合が用いられている．この例では，これらの引数は常に無限リストであり空リストになることはないので，空リストに対する `fmerge` の定義式は用意していない．

型情報についてみてみると，この例では関数 `main` を除くすべてのトップレベル関数に型を明示的に指定しているが，Haskell ではこれらを指定しなくても，処理系内部で行なわれる型推論によって正しく型付け

Main.hs

```
1  module Main where
2
3  import Rama
4
5  main = print (take 3 ramanujan)
```

Rama.hs

```
6  module Rama ( ramanujan ) where
7
8  import Sort ( fsort )
9
10 -- Finding Ramanujan numbers
11
12 ramanujan :: [((Int,Int), (Int,Int))]
13 ramanujan = [(x,y)|(x,y)<-zip s (tail s),
14                    sumcubes x == sumcubes y]
15             where s = fsort sumcubes 1
16
17 sumcubes :: (Int,Int) -> Int
18 sumcubes (a,b) = a^3 + b^3
```

Sort.hs

```
19 module Sort ( fsort ) where
20
21 fsort :: ((Int,Int) -> Int) -> Int
22          -> [(Int,Int)]
23 fsort r k
24   = (k,k) : fmerge r [(k,b)|b<-[k+1..]]
25                      (fsort r (k+1))
26
27 {-
28   Two argument lists should be infinite,
29   so there is no definition for null list.
30 -}
31
32 fmerge :: (a->Int) -> [a] -> [a] -> [a]
33 fmerge r (x:xs) (y:ys)
34   | r x <= r y  = x : fmerge r xs (y:ys)
35   | otherwise   = y : fmerge r (x:xs) ys
```

図 1 Ramanujan 数を求めるプログラム

される．

このプログラムを実行すると，l.15 の式 `fsort sumcubes 1` の結果として

```
[(1,1),(1,2),(2,2),(1,3),(2,3),(3,3),(1,4),
 (2,4),(3,4),(1,5),(4,4),(2,5),(3,5),..]
```

のようなリストが得られ，このリスト中の隣接する 2 要素に対して，3 乗の和が等しくなるものを抜き出すと，Ramanujan 数を整列したものが求められる．

ちなみに，プログラミング言語を最初に説明する際によく用いられる “Hello, World!” を出力するプログラムは，以下の 1 行で記述可能である (`Hello.hs`)．

```
main = putStr "Hello, World!\n"
```

このように `module` 行が省略されたときは，`module Main where` がプログラムの先頭に自動的に補われ，単独で構成される `Main` モジュールとなる．この例では，先の例とは異なり結果の表示に関数 `putStr` を用いている．これは，関数 `print` は多義性をもち，引数が文字列の場合は二重引用符を余計に追加して出力してしまうためである．このように，出力する値が文字列で確定している場合は，関数 `putStr` を用いるほうが自然な出力が得られる．

3 Hugs インタプリタ

3.1 動作環境

Hugs[†2] は，Mark P. Jones らによって開発された Haskell のインタプリタで，現在の最新版は Hugs 98 (2001 年 2 月リリース) である．このシステムは，多くの Unix や Windows 上で動くことが確認されており，コンパイル済のバイナリコードが Win32, Linux, Machintosh の各プラットフォームに用意されている．

Windows 用の Hugs (winhugs) は GUI を備えており，読み込むファイルの指定や動作時のオプション設定なども対話的に行なうことができる．またメニューやツールバーから項目を選択する際に，簡単な説明が最下部に表示されるので，初心者でも扱いやすくなっている．しかし現在の winhugs では，ある種のイベント検出に負荷のかかるポーリングを用いているために，他のプラットフォームと比べて実行効率は低いことに注意されたい．

†2 Hugs 98, `http://haskell.org/hugs/`

```
% hugs
(... 起動メッセージの表示 ...)

Hugs session for:
/usr/local/share/hugs/lib/Prelude.hs
Type :? for help
Prelude> sum [1..10]
55
Prelude> :load Hello.hs
Reading file "Hello.hs":

Hugs session for:
/usr/local/share/hugs/lib/Prelude.hs
Hello.hs
Main> main
Hello, World!

Main> :quit
[Leaving Hugs]
%
```

図 2　hugs セッションの例

3.2　使用法

Hugs を起動するには，hugs コマンドを用いる．図 2 に Hugs でのセッションの一例を示す．なお下線部分がユーザからの入力を示す．

各行で行なっていることは，次の通り．

1　インタプリタの起動

5　組込関数が定義されている Prelude.hs の読み込み

7　入力された式を評価し結果を出力．この例では 1 から 10 までの整数の和を計算．

9　プログラムのロード．プロンプトが読み込んだモジュールの名前に変わる．

15　ロードしたプログラムの実行

18　インタプリタの終了

新しい関数などを定義したい場合は，プロンプト上では定義ができないので，一度ファイルに保存しそれをロードする必要がある．

インタプリタのプロンプトに対して，Haskell の式か : で始まる Hugs コマンドのいずれかが入力可能である．Hugs コマンドの一覧は :? で出力される．

また起動時に引数として与えるオプションの中で，よく利用されそうなものを挙げておく．

+s　式の評価後に，簡約段数と使用セル量を表示する．:set +s コマンドでも設定可能．

+t　式の評価後に，型を表示する．:set +t コマンドでも設定可能．

-h*num*　ヒープサイズを大きくする時に用いる．現在のサイズは :set コマンドで調べられる．

-98　Haskell 98 の言語仕様に加え，Hugs 独自の拡張機能を利用可能にする．

また runhugs コマンドを用いると，対話的ではなくシェルスクリプトのように直接コマンドとして起動できる．このためには，プログラムを以下のような拡張記法 (literate Haskell) で記述し，1 行目に #! に続けて runhugs の絶対パスを記述すればよい．

```
#!/usr/local/bin/runhugs

> main = putStr "Hello, World!\n"
```

literate Haskell とは，Knuth の文芸的プログラミング (literate programming) を Haskell に取り入れた，プログラムとコメントを混在した状態で記述するための記法で，行頭が > で始まっている部分がプログラムとして認識され，それ以外の部分がコメントとして処理される．通常のプログラムと区別するために，拡張子は .lhs が用いられる．

runhugs スクリプトを記述する際に，このような literate Haskell を用いると，このファイルはスクリプトとして実行できるだけでなく，インタプリタでも 1 行目の #! で始まる行がコメントとして扱われるため正しく実行することが可能となる．

3.3　標準ライブラリ

Haskell には，プログラムを作る際に有用となる基礎的な関数が，予め多数用意されている．これらの関数をより多く知っておくことによって，Haskell プログラミングが格段に易しくなる．

これらの標準的に用いられる関数は，Haskell 98 標準ライブラリ [15] として定められており，各処理系は最低限これらの関数を用意しておくことが期待されている．中でも Prelude モジュールにはもっとも基本的な関数が定められており，明示的に import しなくても標準で利用可能となっている．

Hugs には，このようなライブラリ関数に対して，

定義や型などの情報を得るための支援を行なう機能が備わっている．

- 関数の名前がわかっている場合
  `:find` ⟨*func*⟩ コマンドで，指定された関数が定義されているライブラリを表示．表示の際に起動されるエディタは `-E` オプションで変更することが可能．
- モジュールの名前がわかっている場合
  `:browse` ⟨*modules*⟩ コマンドで，指定されたモジュールで定義されている関数やデータ構成子の型情報を一覧として出力する．

これらによってもわからない場合は，直接ライブラリがインストールされたディレクトリ (図 2 の例では/usr/local/share/hugs/lib/) の中を検索するとよい．なお `Prelude.hs` 以外にも，システムには有用なモジュールが多数用意されているので，必要に応じて `:also` コマンドや `import` を使って読み込ませ利用するとよい．

4 GHC コンパイラ

GHC[†3] は，Glasgow 大学で開発された Haskell のコンパイラである．現在の最新版は 5.02 であるが，5.0 系列はまだできて間もないため，OS 毎に用意されているコンパイル済バイナリ (以降パッケージと呼ぶ) が少ないことがある．そのような場合は，1 つ前の版である 4.08.2 を利用するとよい．また 5.0 系列において新たに追加された機能については，4.4 節で解説する．

4.1 動作環境

GHC は大きなコンパイラなので，自力でソースからビルドするよりも，初めはパッケージを入手する方が簡単である．現在以下のプラットフォームに対して 5.02 のパッケージが用意されているので，これらをダウンロードして用いるとよい．

x86 Win32, Linux, FreeBSD

sparc Solaris

もし対象とするプラットフォームに対応するパッケージが用意されていない場合，より多くのパッケージが用意されている 4.0 系列を試すか，ソースから自分でコンパイルする必要がある．GHC をコンパイルするためには，以下のような様々な条件を用意しておく必要がある．

- 100MB 以上の空きディスク
- perl, gcc, Happy などの各種ツール

この中で Happy というツールは，Haskell に対するパーザ生成器であり，C 言語における yacc に相当する．この Happy は Haskell 言語で記述されていることから，Happy をコンパイルするには既に Haskell コンパイラがインストールされている必要があり，このままでは鶏と卵のどちらが先かという問題になってしまう．Happy のパッケージが用意されているプラットフォームは Win32, Linux, FreeBSD, Solaris と GHC より少ないため，通常は，GHC のパッケージをインストールした後に，その GHC を用いて Happy をコンパイルする方が自然であろう．これらを用意するのが難しい場合は，予めパッケージの用意されているプラットフォーム上で GHC を利用する方が無難である．

4.2 使用法

gcc などの C コンパイラと同様に Haskell プログラムを引数に指定し起動すれば，(Haskell の)`Main` モジュールで定義されている `main` 式を実行するような実行モジュール `a.out` が生成される．実行モジュール名を指定する `-o` オプションや，最適化を行なう`-O` オプションも利用可能である．

```
% ghc -O -o main Main.hs
% ./main
```

`--help` オプションをつけて起動すると，よく利用されるオプションについての説明が表示される．ここで示される以外にも GHC では多数のオプションが利用可能なので，より詳細な機能を利用したいときは付属のドキュメントを参照するとよい．

最適化オプションを付けない場合，GHC は内蔵するネイティヴコード生成器 (Native Code Generator, NCG) を用いて，Haskell のコードから高速に直接ア

†3 The Glasgow Haskell Compiler, `http://haskell.org/ghc/`

センブラコードを生成する．一方最適化オプションを付けた場合 NCG は使わずに，一度 C プログラムを生成しさらに外部コマンドである gcc を適用することによって，実行モジュールの生成を行なう．この方がコンパイルに要する時間はかかるものの，gcc における強力な最適化を利用できるため，実行モジュールの効率は良いことが多い．

デバッグなどのために，コンパイルの途中で生成されるファイルを調べたい場合は，C ファイルを生成して止まる `-C` オプション，アセンブラを生成する `-S` オプション，オブジェクトファイルを生成する `-c` オプションを使うとよい．またコンパイル中の詳しい動作内容については `-v` オプションで出力することができる．

4.3 プロファイリング

GHC は，プログラムに内在するボトルネックを調べるために，プロファイリングの機能を備えている．この機能を用いることによって，プログラムの実行時に要した時間と空間を容易に調査することが可能となる．実際に用いるには，コンパイル時と実行時にそれぞれオプションを指定する必要がある．

まずコンパイラを起動する際には `-prof -auto-all` オプションを指定する．`-auto-all` オプションにより，トップレベルで定義されたすべての関数について，その関数が実行時にどれだけヒープを使用したかを記録可能にする．

さらに，実行モジュールを起動する際に `+RTS -p` または `+RTS -h` オプションを用いる．`+RTS -p` オプションを指定した場合，実行後に ⟨実行モジュール名⟩`.prof` というファイルが生成され，時間に関するプロファイリング情報がテキストデータで書き込まれる．また，`+RTS -h` オプションを指定した場合は，実行後に ⟨実行モジュール名⟩`.hp` というファイルが生成される．このファイルから `hp2ps` というユーティリティを用いて PostScript ファイルを生成すれば，使用ヒープ領域をグラフとして出力することが可能になる (図 3)．

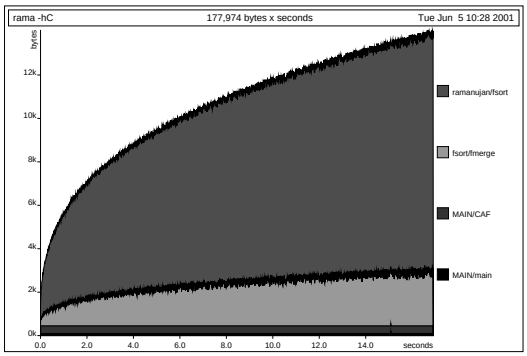

図 3　ヒープ使用量の時間変化

```
% ghci Main.hs
(... 起動メッセージの表示 ...)

Loading package std ... linking ... done.
Compiling Sort              ( Sort.hs, interpreted )
Compiling Rama              ( Rama.hs, interpreted )
Compiling Main              ( Main.hs, interpreted )
Main> :set +s
Main> main
[((1,12),(9,10)),((2,16),(9,15)),((2,24),(18,20))]
(0.10 secs, 2198248 bytes)
Main> :quit
Leaving GHCi.
%
```

図 4　ghci を用いた Ramanujan 数の計算

4.4 GHCi 〜 インタプリタとコンパイラの統合

GHC は 5.0 系列から，従来のコンパイラに加え，インタプリタ (GHCi) としても動作するようになっている．起動するには，シェルスクリプトである `ghci` コマンドを実行すればよく，これは通常の `ghc` コマンドに`--interactive` オプションを付けても同じ動作をする．図 4 に，先の Ramanujan 数を求めるプログラムを `ghci` で実行した例を示す．インタプリタ上でのコマンドは，Hugs と類似するように作られていて`:load`, `:?` などが利用可能である．

またモジュール間の依存関係を自動的に解決する機能も追加された．この機能を用いると，従来は Haskell プログラム間の依存関係を解決するために Makefile を用いていたところが，ルートモジュール (通常は Main) と `--make` オプションを指定するだけで，再コ

表 1 Hugs, GHC の実行効率

	時間 [sec]	ヒープ使用量 [MB]
Hugs	5.27	—
GHCi	0.52	8.48
GHC	0.20	7.28
GHC –O	0.20	6.44

ンパイルが必要なものを自動的に検出できるようになった．これは複数のモジュールに分割して書かれたプログラムをコンパイルする際には，大きな利点となるであろう．

5 Hugs と GHC の比較

5.1 実行効率

表 1 は，Hugs と GHC を用いて Ramanujan 数を小さいものから 10 組求めるプログラムを，PC/AT 互換機 (Pentium II Xeon 450MHz, メモリ 512MB) 上で実行した際に，要した時間とヒープ使用量である．Hugs では，ヒープ使用量はバイト単位ではなく処理系内部で用いられたセル数として表示されるので，ここでは割愛してある．また GHC は，最適化オプションの有無の 2 通りについて試してある．この表によると，概ね GHCi は Hugs の 10 倍速く，コンパイルすればさらに 2 倍以上速くなっている．これより，最新の処理系である GHC 5.0 系ではインタプリタとしての使い勝手も備えた上，効率よく Haskell プログラムを実行することが可能であることから，将来的には Hugs の代わりに広く用いられるものと思われる．

また，従来から Haskell と他の言語処理系の実行効率を比較したものとして文献 [6][8] が知られていたが，最近では，より幅広い言語間の比較に拡張しようという試み [3] が行なわれている．これは，同じ計算機上で同一の問題を解くアルゴリズムを，複数のプログラミング言語で記述することによって比較を可能にしている．扱っている問題は，Ackermann 関数，ヒープソート，行列の積，乱数生成など 25 種類で，対象となる言語は C, C++, Java などの馴染みの深いものから，Lisp, Scheme, SML など歴史のある言語，Perl, Python, Ruby などのスクリプト言語や awk, bash などの UNIX のコマンドインタプリタまで 30 処理系と多岐にわたる．

勿論，すべての問題をすべての言語で記述可能なわけではないが，実験に用いたプログラムと実行した際のログは公開されており，比較結果もグラフ表示されるなど工夫されている．比較の対象となるのは，実行時間，メモリ使用量，プログラムの行数の 3 点についてで，GHC は平均して中の上の位置にランクされている．中でも行数の短さについては，main 関数の引数処理が必要であるような例を除き，かなり上位にランクされており，GHC を用いた Haskell プログラミングの優位性が示されている．詳しい結果については，直接文献 [3] を参照されたい．

5.2 処理系の内部構造

本節では処理系の実装に注目し，インタプリタやコンパイラの内部でどのような過程を経てプログラムの実行が行なわれるかをみることにする．

まず Hugs インタプリタは，プログラムを解釈実行するために抽象機械 G-machine を用いている．抽象機械 (abstract machine) とは，プログラムから実行モジュールを生成する際に，一度抽象的な機械を経由させることによって，元となる言語の詳細とコード生成の過程を分離させることが可能になる技術である．G-machine [1][10] は，非正格な関数型言語を実行するための抽象機械の一種で，グラフ簡約に基づき，その命令列である G-code を解釈実行する．すなわち Hugs では，読み込んだ Haskell プログラムをまず G-code に変換し，その後内部に実装されている G-machine が G-code を解釈実行する流れになっている．

一方 GHC では，内部的に Core 言語と STG 言語 (Spineless Tagless G-machine [13] 上の言語) を経由した後，C プログラムを生成し，最終的に外部の C コンパイラを呼び出す構成になっている (図 5 参照)．Core 言語と STG 言語は，G-code のように抽象機械の命令列で表わされるものとは異なり，それだけで小さな関数型言語を構成している．このため Core 言語/ STG 言語上の最適化処理はプログラム変換とみなすことも可能 [14] で，状態遷移を用いている通常

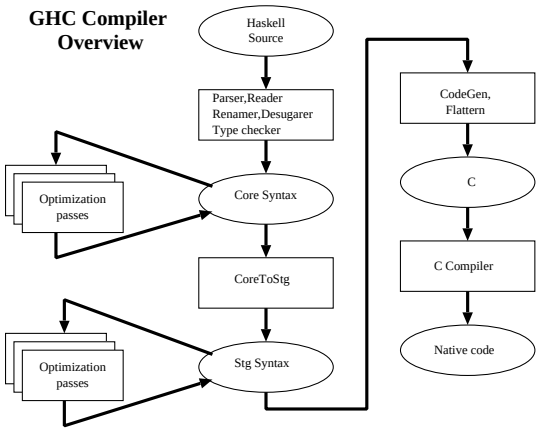

図 5 GHC の内部構造

のコンパイラと比べて，ユーザが任意のプログラム変換をコンパイラ内に追加実装しやすくなっている．その一例として，Core 言語上に融合変換 (program fusion) の処理を実装し，その有効性を検証する試みも行なわれている [11] [12]．なお GHC の内部構造についてさらに詳しく知るには，文献 [4] [5] を参照するとよい．

6 おわりに

今後，関数プログラミングが普及するために必要な要因として，文献 [18] では以下の 3 つが挙げられていた．

1. 効率的な実行システム
   従来は，標準的なノイマン型計算機上で，効率的に実行可能なシステムがなかった．
2. 関数プログラミングの実践
   トイプログラムでの議論に終わることが多く，実用的なプログラムに用いられる例が少ない．
3. プログラム運算 (calculation)
   数式の変形と同様に，プログラムの変形によって新たなアルゴリズムを導出できることがわかってきた．

その後研究が進み，1 については他の手続き型言語などと比べて遜色ないほどのレベルまで来ていることが窺える．また 2 に関しては定理証明系などの分野で活用されるようになり，3 についても徐々に新しい提案がなされてきているので，今後はさらなる発展を望みたい．

参考文献

[1] Augustsson, A: A Compiler for Lazy ML, *Proc. of the ACM Symposium on Lisp and Functional Programming,* ACM Press, 1984, pp.218–227.
[2] Bird, R.: *Introduction to Functional Programming using Haskell, 2nd edition,* Prentice Hall Press, 1998,
[3] Bagley, D.: The Great Computer Language Shootout, `http://www.bagley.org/~doug/shootout/`
[4] Chakravarty, M.: The Glasgow Haskell Compiler Commentary. `http://www.cse.unsw.edu.au/~chak/haskell/ghc/comm/`.
[5] Chitil, O.: Adding an Optimisation Pass to the Glasgow Haskell Compiler, unpublished, 1997. `http:// www-users.cs.york.ac.uk/~olaf/PUBLICATIONS/extendGHC.ps.gz`.
[6] Hartel, P. H. et al.: Benchmarking Implementations of Functional Languages with “Pseudoknot”, a Float-Intensive Benchmark, *J. Functional Programming*, Vol.6, No.4 (1996), pp.621–655.
[7] Hudak, P.: Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Survey,* Vol.21, No.3 (1989), pp.359–411. (邦訳 武市正人: 関数プログラム言語の概念・発展・応用, コンピュータ・サイエンス ’89, 共立出版, 1991, pp.37–87.)
[8] Hudak, P. and Jones, M. P.: Haskell vs. Ada vs. C++ vs. Awk vs. ..., An Experiment in Software Prototyping Productivity, unpublished, 1994. `http://haskell.org/papers/NSWC/jfp.ps`.
[9] Hudak, P., Peterson, J. and Fasel, J.: A Gentle Introduction to Haskell, `http://haskell.org/tutorial/`.
[10] Johnsson, T.: Efficient Compilation of Lazy Evaluation, *Proc. SIGPLAN ’84 Symposium on Compiler Construction*, ACM Press, 1984, pp.58–69.
[11] 尾上能之, 胡振江, 武市正人: HYLO システムによるプログラム融合変換の実現, コンピュータソフトウェア, Vol.15, No.6 (1998), pp.52–56.
[12] 尾上能之, 胡振江, 岩崎英哉, 武市正人: プログラム融合変換の実用的有効性の検証, コンピュータソフトウェア, Vol.17, No.3 (2000), pp.81–85.
[13] Peyton Jones, S. L.: Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, *J. Functional Programming*, Vol.2, No.2 (1992), pp.127–202.
[14] Peyton Jones, S. L. and Santos, A: A transformation-based optimiser for Haskell, *Science of Computer Programming*, Vol.32, No.1–3 (1998), pp.3–47.
[15] Peyton Jones, S. L. and Hughes, J. (eds.): Standard libraries for the Haskell 98 Programming

Language, *Department of Computer Science Tech Report YALEU/DCS/RR-1105, Yale University*, 1999.
[16] Peyton Jones, S. L. and Hughes, J. (eds.): Report on the Programming Language Haskell 98, *Department of Computer Science Tech Report YALEU/DCS/RR-1106, Yale University*, 1999.
[17] 武市正人: 関数プログラミングの実際, コンピュータソフトウェア, Vol.9, No.1 (1991), pp.3–11.
[18] 武市正人: 特集「いま欲しいブレークスルー」関数プログラミング, bit, Vol.31, No.3 (1999), pp.7–10.
[19] Thompson, S.: *Haskell: The Craft of Functional Programming, 2nd edition,* Addison-Wesley, 1999.
[20] Turner, D. A.: Miranda: a non-strict functional language with polymorphic types, *Functional Programming Languages and Computer Architecture,* Springer-Verlag, LNCS 201, 1985, pp.1–16.