解　説

プログラミング言語最新情報　I

ML
——多相型システムをもつ関数型言語†——

大　堀　　淳††

1. はじめに

ML は，パターンマッチング，例外処理，参照型，モジュールシステムなどの機能を関数型言語の中にうまく統合した汎用プログラミング言語である．その静的型システムは，プログラムのもつ汎用性を正確に反映した多相型を自動的に推論することができ，エラーの少ないプログラムを効率良く開発することを可能にしている．最近の実装技術の蓄積によって実用的なコンパイラも開発されている．ML はまた，その意味論が厳密に定義され，型システムの安全性などの望ましい性質が理論的に検証されている数少ない言語である．この厳密性は，ML で書かれたプログラムに高い信頼性を与えるとともに，プログラミング言語の種種の新しい機能の研究の基礎ともなっている．

本稿では，汎用プログラミング言語としての ML とプログラミング言語の研究の対象としての ML の二つの側面を紹介する．まず **2.** で ML の歴史的な発展を概観した後，**3.** で ML の特徴を例を用いて説明する．**4.** では ML の基礎理論を説明し，**5.** で ML の応用事例を紹介する．さらに **6.** では現在の ML に関連した研究の動向を紹介する．

2. ML 発展の歴史

ML は，1970 年代の後半に Edinburgh 大学で，証明導出システム Edinburgh LCF[12] のための記述言語として開発された．LCF システムの記述の対象は定理や公理，推論規則などである．ML はそれらを計算機システムの中で表現，操作するための言語，すなわちそれら数学的対象が属する形式言語のメタ言語 (Meta Language) であった．ML の名前はこの歴史的な事実による．名前のみならず，ML の特徴の多くは LCF のメタ言語としての要求に由来している．

LCF システムの重要な構成要素の一つは，公理や推論規則を合成し効率良く証明を導出するための証明戦略 (proof tactics) である．プログラミングの観点からは，推論規則は命題から命題を導出する関数とみなされ，したがってそれらを操作する証明戦略は，関数を合成し新たな関数を生成するような高階の関数 (higher-order functions) とみなすのが最も自然である．そこで LCF ML は，証明戦略などがうまく表現可能なように，高階の関数を自由に値として扱うことができる関数型言語として設計された．関数型言語ではあるが，大規模システムを効率良く構築するために，例外処理や参照型などの非関数的な機能も含んでいた．

LCF システムに限らず種々の証明導出システムや推論システムは，以上のような理由から，Lisp や Scheme などの関数型言語で書かれていることが多い．それら既存の言語と比較した LCF ML の最大の特徴は，プログラムの型を自動的に推論する静的型システムを装備していたことである．証明戦略などは通常，種々の場合に適用可能な一般性のあるスキーマであり，それらの表現も，その一般性を反映した汎用性ある関数であることが望ましい．一方，大規模なシステムの信頼性の向上のためには，複雑なプログラムに潜在する誤りの多くをコンパイル段階で検出することができる静的型チェックも強く望まれる．LCF ML の型システムの設計者であった Robin Milner は，複雑なプログラムに潜む不整合をコンパイル段階で自動検出することを目的として，高階の関数を含

† ML—a Functional Language with a Polymorphic Type System—by Atsushi OHORI (Research Institute for Mathematical Sciences, Kyoto University).

†† 京都大学数理解析研究所

☆ 本稿のもとになった研究の大部分は，筆者が沖電気工業(株)に勤務していたときになされたものである．

むプログラムの最も一般的な型を自動的に推論する型推論 (type inference) の理論とその実用的アルゴリズムを構築した[28]. この理論は，型宣言を一切必要とせずに，汎用性ある関数の定義と静的な型チェックによる型の不整合の検出とを同時に実現する画期的なものであった. LCF ML は，この型推論理論を基礎に設計された型システムによって，型付き言語の安全性と信頼性を損なうことなく Lisp などの型なし言語（動的に型付けられた言語）の柔軟性を実現し，LCF システムのプログラミングの信頼性と柔軟性の向上に大きく貢献した.

LCF ML の成功は多くのプログラミング言語の研究者の注目を集めた. 特にその型システムは，証明導出システムのメタ言語としてばかりでなく，汎用のプログラミング言語にとっても有用性が高いと認識され，それ以後，LCF ML を拡張し汎用プログラミング言語を設計する努力が行われた. それらの中で今日の ML* に採用されている重要な成果は，パターンマッチングとモジュールシステムの導入である.

パターンマッチングは，処理したいデータ構造をパターンを用いて記述することによって，複雑なデータの操作の記述を容易にする機構である. この機構は，Edinburgh 大学で 1970 年代に開発されたプログラミング言語 HOPE[6] で提案された. Milner はこのパターンマッチング機構を LCF ML の多相型システムと統合し，現在の ML の骨格となるプログラミング言語 The Standard ML Core Language を提案した[29]. HOPE 言語にはまた，モジュールシステムも含まれていた. MacQueen は，HOPE 言語にあったモジュールの考えを一般化し，モジュールのための型システムを構築し[26]，それを基に ML のためのモジュールシステムを設計した[23]. 論文[23),29)]はその後改訂され，Harper によるストリームに基づく入出力処理の記述とともに，当時，LCF や ML および HOPE 言語などに興味をもつ研究者たちの非公式な会報であった *Polymorphism* にまとめて発表された[13),24),30)]. この三つのレポートが今日の Standard ML の原型である. その後以上の提案に対して注意深い検討と改訂およびその整合性の理論的な検証がなされ，1990 年 ML 言語の定義 The Definition of Standard ML[31] が出版された.

Standard ML の仕様決定およびその理論的検証の努力と並行して，ML の処理系の実装の努力が行われた. LCF ML 以後の，独立したプログラミング言語としての ML コンパイラの最初のものは，恐らく Cardelli のシステムであろう[7]. このコンパイラは 1980 年代の後半に Edinburgh 大のグループによって，上記の三つのレポートに定義された仕様をほぼ満たす Standard ML コンパイラに改良され，大学や研究機関などで使用された. また Cambridge 大学やフランスの INRIA 研究所でもそれぞれ独立に ML の処理系が作成された. 最近では，AT&T の MacQueen と Princeton 大学の Appel が，Standard ML of New Jersey コンパイラを完成させた. このコンパイラは効率良いコードを生成する堅牢なシステムであり，実用言語としての Standard ML の普及に大きく貢献した. それ以外にも現在複数の処理系が作成されている.

3. Standard ML の特徴

本章では，ML の特徴を例を用いて簡単に説明する.

3.1 対話型プログラミング

ML のプログラムは一つの式である. 複雑で大きなプログラムも，多数の式の組合せによって構成される一つの式である. プログラムの実行は，式が示す値を計算することによって行われる. ML でのプログラミングは，Lisp などの対話型言語同様，以下の手順を繰り返すことによって行われる.

1) ユーザによる式の入力.
2) システムによる型検査，コンパイル，実行.
3) 実行結果とその型情報の表示.

以下に簡単な対話型プログラミングの例を示す.

```
- 113;
113: int*
- (2*4)*(3+5)*3+(10-7);
195: int
```

☆ Milner の多相型システムは，Miranda[49] や Lazy ML[3], Haskell[19] などの遅延評価に基づく言語にも採用されている. しかし，遅延評価に基づく言語には，LCF ML に最初から含まれている参照型や例外処理などの非関数的機能の導入は難しい. 主にこの理由から通常 ML と言う場合は，作用順評価に基づく言語をさす. 本稿でもこの見方に従い，Standard ML を ML の標準とみなすことにする.

☆ 本稿では，ユーザの入力とシステムの出力を区別するために，後者には *slanted font* を使用する.

```
- val dollar = 108;
val dollar = 108: int
- 400 * dollar;
43200: int
```

ユーザの入力行の先頭の“-”は，MLシステムが表示する入力促進，最後の“;”は入力の終りの指示である．val $x=E$ は，式 E の結果に x という名前をつけ，後の式で使用するための構文である．

3.2 高階の関数と多相関数

プログラミング言語は，単に計算手順の記述の手段に留まらず，プログラムを構築する上での抽象化の概念と構造化の機構を提供する．関数型言語であるMLでは，高階の関数の定義とその利用に関する柔軟で強力な機構を提供することによってこの目的を果たしている．

関数は fn x => E の形をした式で定義する．この式は，x を引数として受けとり，x を含む式 E の値を計算する（名前のない）関数を表す．以下に簡単な例を示す．

```
- val double = (fn x => x * 2);
val double = fn: int -> int
- double 3;
6: int
```

式の値が関数である場合，評価の結果は単に fn と表示される．型情報 int -> int は，式の評価の結果が，int 型の引数を受けとり int 型の値を計算する関数であることを表す．double 3 は関数 double の 3 への関数適用である．ML では，関数適用をこのように単に関数と引数を並べて書く．

以上の関数定義と名前の定義が統合されたものが以下の例で示す fun 宣言文である．

```
- fun duplicate x = x^x;
val duplicate = fn: string -> string
- duplicate "ML";
"MLML": string
```

duplicate 関数の中で使われている“^”は文字列の連結演算子である．

汎用性のある高階の関数も定義できる．たとえば，与えられた関数を与えられた値に 2 回適用する関数 twice を以下のように定義できる．

```
- fun twice f x = f (f x);
val twice = fn: ('a -> 'a) -> 'a -> 'a
```

推論された型情報 ('a -> 'a) -> 'a -> 'a は ('a -> 'a) -> ('a -> 'a) の括弧が省略されたものである．これは twice が，'a -> 'a 型の関数を引き数として受けとり，'a -> 'a 型の関数を結果として返す高階の関数であることを示している．'a は任意の型を代表する型変数である．twice は，この型変数を適当な型で置き換えて得られる任意の型の関数として使用できる．このように型変数を含む型を多相型 (polymorphic type) と呼び，多相型をもつ関数を多相関数と呼ぶ．

多相型の中の型変数の置き換えは，その多相関数の使用時にシステムによって自動的に行われる．以下に twice の使用例を示す．

```
- twice double 2;
8: int
- val fourtimes = twice double;
val fourtimes = fn: int -> int
- fourtimes 3;
12: int
```

twice double 2 は ((twice double) 2) の括弧が省略されたものである．これは，twice を関数 double に適用し，その結果として得られる関数を値 2 に適用する二重の関数適用である．この適用に先だって，型変数 'a が int に置き換えられ，twice は (int -> int) -> (int -> int) 型の関数として使用されている．それに続く例では，twice double の結果得られる関数を独立の値として使用している．このように ML では，関数を，整数などのデータと同様に値として使用することができる．

以下の例から分かるように，型変数は，引き数の型に応じて自動的に種々の型に置き換えられる．

```
- twice duplicate "ML";
"MLMLMLML": string
- val fourth = twice twice;
val fourth = fn: ('a -> 'a) -> 'a -> 'a
- fourth double 2;
32: int
```

1 番目の例では，'a は sring 型に置き換えられている．twice twice は twice の自分自身への適用であり，結果は関数を 2^2 回適用する高階の関数である．この例では，'a は多相型 'a -> 'a 型に置き換えられ，その結果も多相型の関数である．

ML では，以上のような多相型高階関数をうまく使うことによって，複雑なプログラムを簡潔に

しかも安全に記述することができる.

3.3 データ型の定義とパターンマッチング

ML のもう一つの特徴は，多相関数の定義機構が，ユーザ定義のデータ型およびパターンマッチングを通じたデータ型の利用機構とうまく統合されている点である.

データ構造の定義は，そのもち得る内部構造をタグを付けて列挙する datatype 文を用いて行う. たとえばノードのデータ型が 'a である2分木 'a tree は以下のように定義できる.

```
- datatype 'a tree =
    Empty
  | Node of ('a * 'a tree * 'a tree);
  datatype 'a tree
  con Empty: 'a tree
  con Node: ('a * 'a tree * 'a tree) -> 'a tree
```

* は組型構成子である. "|" は「または」の意味であり，この宣言は，型 'a tree を，タグ Empty のみのデータかまたは，Node というタグをもつ ('a * 'a tree * 'a tree) 型のデータと定義している. 定義に使用されている型変数 'a は，2分木型構成子 tree のパラメータであり，使用されるデータの型に応じて自動的に適当な型に置き換えられる. タグ Empty と Node は2分木データの生成子として使用される. 以下に簡単な例を示す.

```
- val T = Node (1, Empty, Empty);
val T = Node (1, Empty, Empty): int tree
```

この例の場合，Node への第一引数が整数であるため，型変数 'a が自動的に int に置き換えられ，結果の型は int tree となっている.

datatype 文で定義されたデータ構造はパターンマッチングを通じて利用する. パターンマッチングは，関数の引数などに，変数の代わりに，処理したいデータ構造を表すパターンを記述することを許す機構である. たとえば，木の高さを計算する関数は，パターンを用いて以下のように簡潔に書ける.

```
- fun height Empty = 0
    | height (Node(x, L, R)) =
        1+max(height L, height R);
val height = fn: 'a tree -> int
```

height は，引数が Empty なら0を，Node(x, L, R) の形をした木なら，1+max(height L, height R) を計算する関数である. ここで引数として書かれた Empty と (Node(x, L, R)) がパターンである. このようにパターンを記述することによって，データ構造のテストとその分解とが自動的に行われる. さらに，このパターンマッチングの仕組みは型推論システムとエレガントに統合されている. たとえば height 関数に対して，ML システムはその汎用性を正確に反映した多相型を推論している.

3.4 例外処理

純粋な関数型言語では，広域的なジャンプや変数の破壊的代入は表現できない. しかしこれら機能があるとプログラムの効率や読みやすさが大幅に向上する場合がある. ML は，その静的型システムを拡張し例外処理と参照型を加えることによって，関数型言語の利点を損なうことなくこれら二つの機能を導入することに成功している. ここでは例外処理のみを簡単に説明する.

上にあげた tree 型を使って，int データをキーとし，string 型データを検索する2分探索木を作るとする. 探索木は (int * string) tree の型のデータで表現できる. 探索関数 find を，探索木とキーの値の組を引数としてとり，結果の値を返す関数として定義したい. しかしキーの値が存在しない場合は返す値がない. そのような場合を例外として処理できれば，全体としてより簡潔なプログラムになることが多い. 例外は exception 文で宣言した後，raise 文で発生できる. 上記の探索関数は，例外の識別子として Find を用いて，以下のように定義できる.

```
- exception Find;
- fun find (Empty, key : int*) = raise Find
    | find (Node((k, info), t1, t2), key) =
        if key = k then info
        else if key < k then find (t2, key)
        else find (t1, key);
val find =
  fn: ((int*string)tree*int) -> string
```

発生した例外は handle 文で処理する. たとえば郵便番号と都市名を格納した探索木 zip_code_data が作成済みとすると，与えられた郵便番号から都市名を探索する関数は以下のように定義できる.

```
- fun city_name zip =
```

* Key : int は，引数 key の型が int であることの宣言である. 関数の中で使用している比較演算子 = およびくが多重定義された演算子であるため必要となる.

```
    find (zip_code_data, zip)
    handle Find =>
      "illeagal post code";
city_name = fn: int -> string
```

もし与えられた郵便番号が zip_code_data データベースに存在しない場合は，find 関数が Find 例外を生成し，その例外が city_name 関数の handle 文で処理され，"illeagal post code" が返される．

3.5 モジュールシステム

Standard ML は，多相型システムと統合された柔軟で強力なモジュールシステムを提供している．ML の多相型システムは，モジュールの定義とその利用機構に拡張されており，モジュールを含んだプログラムの型の整合性も，ML の型システムで完全にチェックされる．このモジュールシステムは，大規模なシステムを安全にしかも効率良く開発する上で有効であり，実用言語としての Standard ML の大きな強みの一つである．

モジュールは種々の定義の集まりである．モジュールに含めることのできる定義は，値の定義，関数定義，データ型の定義，例外定義，およびモジュール定義などである．モジュールの中にさらにモジュールを定義できるため，種々の資源を階層化し，大規模なシステムを構造化することが可能である．

モジュールは structure 文で定義する．たとえば，上記の二分木の定義と探索関数をまとめて，**図-1** のようなモジュールを定義できる．モジュール内の資源は

module_name・*name*

の形で参照し使用できる．また，open 文

open *module_name*

によって，モジュール内のすべての名前を直接参照可能とすることもできる．

以上のような単純なモジュールに加えて，モジュールを引数として受けとりそれを使って別のモジュールを生成するモジュールも定義することができる．またモジュール内の特定の名前のみを利用可能にすることもできる．

```
structure Search = struct
  datatype 'a tree =
      Empty
    | Node of 'a * 'a tree * 'a tree
  exception Find
  fun find (Empty, key) =
      ...
end
```

図-1 モジュール定義の例

4. ML の理論的基礎

ML も他のプログラミング言語同様，多くの機能を含んだ複雑なシステムであり，その仕様は設計者らの注意深い分析と検討の結果決定された．しかも ML の場合，それらの分析は形式的枠組の中で厳密に行われた．決定された ML の定義[31]は，だれが読んでも同一の意味をもつ厳密な数学的言葉で書かれている．この厳密な定義によって多くの望ましい性質が検証されている．たとえば，ML は安全な言語であることが保証されている．つまり，ML で書かれたプログラムは，実行時に未定義の操作を実行して予期せずに停止するようなことはおこらないのである．ML の厳密な理論的基礎の中核が ML の型システムの理論である．

プログラミング言語が提供するプログラムの抽象化の概念や構造化の機構の重要な部分は，その言語の型システムによって規定される．型システムは，言語で表現可能な式がどのような型をもち得るかを規定する形式系のことである．この形式系において中心となる概念は，ある式がある条件のもとである型をもつことを表す型判定である．ML ではさらに，この型判定を自動的に推論する機構，すなわち型推論アルゴリズムが備わっている．本章では ML の型システムの定義とその型推論アルゴリズムの概要を紹介する．

ML の核言語の型システムの性質の大部分は，以下の構文規則で与えられる式のもつ型の性質を通じて分析することができる．

e ::= c | x | fn x => e | e e |
let x = e in e

"|" で区切られた各式は，メタ変数 e で代表される集合を決定する生成規則と解釈する．たとえば，fn x => e は，「もし e が式なら，fn x => e も式である」という生成規則を意味する．

c はプログラムで使われる定数の集合を，x は変数の集合を代表する．すでに前章で説明したように，fn x => e は関数定義，e e は関数適用を表す．let x = e_1 in e_2 は，多相型をもつ式 e_1 に x という名前をつけ，プログラム e_2 の中で使用するための構文である．ML での名前の定義 val x = E;

とそれに続く文脈 E' でのその利用は，式 let $x = E$ in E' でモデル化される．fun 宣言による多相型関数の定義とその利用は，この let 式と関数定義のための fn 式の組合せで実現できる*.

以上定義した式の型システムを定義する．理解を容易にするために，let構文を含まない式の型システムをまず定義し，後にそれを let 構文に拡張する．let構文を含まない式のとり得る型の集合は以下の構文で与えられる．

$\tau ::= t \mid b \mid \tau \rightarrow \tau$

t は，ML システムで 'a のように表示される型変数集合，b は int などの型定数集合を表す．

式は一般に変数を含んでいるため，その型は変数のもつ型に依存する．変数にその型を与える型環境 $\mathcal{A}$ を変数の部分集合から型集合への関数とする．型システムは通常「式 e が，変数の型付けを行う型環境 $\mathcal{A}$ のもとで型 τ をもつ」という性質を意味する以下の形の型判定の導出システムとして定義される．

$\mathcal{A} \triangleright e : \tau$

型判定導出システムは，一般に以下のような型付け規則の集合で与えられる．

$$\frac{\mathcal{A}_1 \triangleright e_1 : \tau_1 \quad \cdots \quad \mathcal{A}_k \triangleright e_k : \tau_k}{\mathcal{A} \triangleright e : \tau}$$

この規則は，「各式 e_i が型環境 $\mathcal{A}_i$ のもとで型 τ_i をもつならば，式 e は型環境 $\mathcal{A}$ のもとで型 τ をもつ」ということを表す推論規則である．上段の条件のない規則は常に成り立つ型判定を表し，導出システムの公理に相当する．

let 構文を含まない ML 式に対する型付け規則を**図-2** に示す．公理 CONST は，定数 c は常にそのあらかじめ定められた型をもつ式として使用できることを示す．公理 VAR は，変数 x がすでに型環境 $\mathcal{A}$ で $x : \tau$ と定義されていれば型 τ をもつ式として使用できることを示す．たとえば型判定

$\{x : \tau\} \triangleright x : \tau$

CONST　$\mathcal{A} \triangleright c : \tau$ if c has the type τ

VAR　$\mathcal{A} \triangleright x : \tau$ if $\mathcal{A}(x) = \tau$

ABS　$\dfrac{\mathcal{A}\{x : \tau_1\} \triangleright e_1 : \tau_2}{\mathcal{A} \triangleright \text{fn } x \Rightarrow e_1 : \tau_1 \rightarrow \tau_2}$

APP　$\dfrac{\mathcal{A} \triangleright e_1 : \tau_1 \rightarrow \tau_2 \quad \mathcal{A} \triangleright e_2 : \tau_1}{\mathcal{A} \triangleright e_1\ e_2 : \tau_2}$

図-2　ML の核言語の型システム(1)

☆ 正確には，再帰的定義のための機構が必要であるがここでは省略した．

は任意の τ に対して常に成り立つ．規則 ABS は，関数式 fn $x \Rightarrow e_1$ が型環境 $\mathcal{A}$ のもとで型 $\tau_1 \rightarrow \tau_2$ をもつのは，式 e_1 が型環境 $\mathcal{A}\{x : \tau_1\}$ のもとで型 τ_2 をもつときであることを示している．ここでの記法 $\mathcal{A}\{x : \tau\}$ は，$\mathcal{A}'(x) = \tau$ かつ，$x \neq y$ である任意の $y \in domain(\mathcal{A})$ について $\mathcal{A}'(y) = \mathcal{A}(y)$ を満たす型環境 $\mathcal{A}'$ を表す．型環境の変更 $\mathcal{A}\{x : \tau_1\}$ によって，通常のプログラミング言語における変数の静的なスコープ規則が表現されていることに注意．規則 APP は，関数適用に関する型チェック手続きの自然な表現である．

関数式で引数の型が指定されていないため，与えられた式は一般に複数の型をもち得る．たとえば任意の型 τ に対して型判定 $\{x : \tau\} \triangleright x : \tau$ が成り立つから，任意の型 τ について以下の型判定が成り立つ．

$\emptyset \triangleright \text{fn } x \Rightarrow x : \tau \rightarrow \tau$

式が一つ以上の型判定をもつ場合は，その式は常に，主要な型判定 (principal typing) と呼ばれる特に重要な型判定をもつ．

定理 1　与えられた式 e が型判定をもてば，主要な型判定 $\mathcal{A} \triangleright e : \tau$ が存在し，e に対するその他の可能な型判定はすべてその主要な型判定の型変数を適当な型で置き換え，必要に応じて型環境に前提を付け加えることによって得ることができる．■

ある式の主要な型判定は，その式がもつすべての型判定を代表する型判定である．たとえば，

$\emptyset \triangleright \text{fn } x \Rightarrow x : t \rightarrow t$

は式 fn $x \Rightarrow x$ の主要な型判定であり，この式のもつすべての型判定は，この主要な型判定の型変数 t を適当な型で置き換え，(不必要な) 型宣言を型環境に追加することによって得ることができる．さらに重要なことに，主要な型判定を自動的に計算するアルゴリズムが存在する．

定理 2　任意の式 e に対して，もし e が型をもてばその主要な型判定を計算し，もし型をもたなければエラーを報告するアルゴリズムが存在する．■

アルゴリズムのおおよその構造は以下のとおりである．

1)　自分自身を再帰的に用いて，式を構成する部分式の型判定を計算する．

2)　式の構造に対応する型付け規則に従い，部分式の型判定の間に成り立つべき条件を等式の集

合として求める.

3) 単一化アルゴリズムを用いて，等式集合を満たす最も一般的な解を，型変数の置換として求め，その置換を適用し型判定を求める.

例として，式 f x の主要な型判定を計算する過程を考えてみよう．まず，部分式 f と x の主要な型判定を計算する.

$\{f: t_1\} \triangleright f: t_1$

$\{x: t_2\} \triangleright x: t_2$

求める式 f x は関数適用であり，その型判定を導出する規則は APP である．そこで，以上二つの型判定から，規則 APP の二つの前提に当てはまる最も一般的な型判定を作ることを考える．まず型環境が共通でなければならないことから，以下の型判定を得る.

$\{f: t_1, x: t_2\} \triangleright f: t_1$

$\{f: t_1, x: t_2\} \triangleright x: t_2$

さらに，規則 APP の前提となる型判定の型の関係から，t_1, t_2 の間には，ある型 τ に対して

$$t_1 = t_2 \to \tau$$

の関係が成り立たなければならない．この条件のもとで，式 f x の結果の型は τ となる．この条件を満たす最も一般的な解は，t_3 を新しい型変数として，t_1 を $t_2 \to t_3$ に置き換える置換である．以上から，式 f x の主要な型判定が

$\{f: t_2 \to t_3, x: t_2\} \triangleright f\ x: t_3$

と求まる.

以上の性質は Hindley によってラムダ計算と等価なコンビネータ論理の分野で発見されていた[17]．ML の型システムの基礎となった Milner の型推論システムは，以上の型推論システムを，多相型をもつ式の利用機構である let 宣言に拡張したものである．その型理論的形式化は Damas と Milner によって与えられた．彼らはまず，今まで使用した型集合に加えて，以下の生成規則で与えられる多相型の集合を定義した.

$$\sigma ::= \tau \mid \forall t.\sigma$$

多相型 $\forall t.\sigma$ は，σ の中に現れる型変数 t を適当な型で置き換えて得られる種々の型の式として使用可能な汎用性ある式の型である．多相型をもつ式の定義とその利用に関する規則は**図-3** で与えられる．規則 GEN は，型環境の中で特に仮定されていない型変数を抽象化して，多相型をもつ式を作る規則である．規則 INST は，式のもつ多相型の中の型変数を適当な型で置き換え，種々の型の式として使用可能にする規則である．ここで使用されている記法 $\sigma[\tau/t]$ は，σ の中の型変数 t を τ で置き換えて得られる型を表す．規則 LET は，多相型をもつ式 e に名前 x をつけ，それを種々の型の式として使用するための規則である.

$$\text{GEN}\quad \frac{\mathcal{A} \triangleright e: \sigma}{\mathcal{A} \triangleright e: \forall t.\sigma}\ t \text{ not free in } \mathcal{A}$$

$$\text{INST}\quad \frac{\mathcal{A} \triangleright e: \forall t.\sigma}{\mathcal{A} \triangleright e: \sigma[\tau/t]}$$

$$\text{LET}\quad \frac{\mathcal{A} \triangleright e_1: \sigma \quad \mathcal{A}\{x: \sigma\} \triangleright e_2: \tau}{\mathcal{A} \triangleright \text{let } x = e_1 \text{ in } e_2: \tau}$$

図-3 ML の核言語の型システム(2)

以上の拡張によって，多相型をもつプログラムの定義と使用が可能になる．たとえば

```
let ID = fn x => x in
    ··· (ID 3) ··· (ID "ML") ···
```

の形をした式において，ID の型は，式 fn x => x のもつ主要な型 $t \to t$ の型変数を抽象して得られる多相型 $\forall t.t \to t$ となり，ID 3 では t を int で置き換えたi nt -> int 型の関数として，また ID "ML" では，t を string で置き換えて得られる string -> string 型の関数として使われている.

let x = e_1 in e_2 式の型推論は以下のように行なえば良い.

1) まず e_1 の主要な τ 型を求め，その中の型変数の中で型環境に現れないものを $\forall$ で束縛し，それを x の型 σ として記録する.

2) e_2 の型推論を以前同様に行う．ただし，e_2 の中で x が現れたら，その型は，ステップ 1) で記録した型 σ の中の型変数のうち，$\forall$ で束縛された型変数をすべて新しい型変数で置き換えて得られる型として推論を進める.

先に説明した型推論アルゴリズムに，以上の let 式の型推論を加えたものが，Milner による ML の型推論アルゴリズム $\mathcal{W}$ である．そのアルゴリズムは，本章で定義した Damas と Milner による多相型システムに関して完全であること，すなわち，アルゴリズムは常にプログラムのもつ主要な型を推論することが証明されている.

5. ML の応用事例

ML は伝統的な定理証明システムの構築に使用されてきた．Edinburgh LCF システムにはじまり，Edinburgh LCF を発展させた Cambridge LCF

システム[39], Gordon による VLSI の検証のための HOL システム[11], Constable らによる直観主義的型理論に基づく定理証明システム Nuprl[8], Paulson による種々の証明チェッカのためのプラットフォーム ISABELLE[40] など多くのシステムが ML で実装されている．以上は，大学で研究用に作られた実験システムとしての性格が強い．

5.1 Standard ML of New Jersey コンパイラ

ML を使用しての実用システム構築の報告は，ML の実用コンパイラの普及が始まったばかりであることもあり，あまり多くは存在しないようである．筆者の知る出版された唯一のものは，MacQueen と Appel による Standard ML で書かれた Standard ML of New Jersey コンパイラの構築報告[1),2)]である．彼らの初期の意図は種々の実験を行うためのインタプリタを作ることであったが，その後このプロジェクトはその目的を拡大し，Standard ML の高品質の実用のコンパイラを構築することを目指した．後期の論文[2)]で彼らはその目的について「Standard ML はこれまでに作られた中で最良の汎用プログラミング言語の一つでありうる．それを証明するために高品質，堅牢で高速なコンパイラを提供する」と述べている．

このコンパイラは約3万行のソースコードで作られている．このうち，カーベジコレクションを含んだ実行時ルーチンと機械種別ごとの機械命令の記述，およびオペレーティングシステムのインタフェースルーチンはC言語で書かれているが，その他の 90% 以上は ML で書かれている．その構造は，入力したプログラムを簡単なラムダ式に変換する部分と，そのラムダ式を機械命令に変換する部分に大別される．前半では，入力プログラムを構文解析しそれに対応する抽象構文木を生成し，本稿 **4.** で紹介したような型推論を行った後，パターンマッチング処理をラムダ式に変換する．後半では，ラムダ式を CPS 式と呼ばれる機械命令に近い中間語に変換し，機械の種類に依存しない最適化を行った後，機械種類に応じた機械命令を生成する．以上の処理の各段階ごとに生成されるデータは ML の datatype 文で定義したデータ構造で表現され，各段階の処理は ML のモジュールとして実現されている．この構造のため，比較的大規模なシステムにも関わらず，作成者本人以外にも分かりやすく改良や改造が行いやすい構造になっている．

完成されたコンパイラはほぼ彼らの目的を満たすものになっているといえよう．彼らの報告によれば，たとえば MIPS プロセッサ上での Knuth-Bendix ベンチマークの実行時間は，最適化オプションつきでコンパイルしたCプログラムの 1.6 倍程度に収まっている．現在実装技術の研究は急速に進歩しており，現在の速度はこの数字より改善されていると思われる．この結果は，Standard ML が単に大学での実験研究用の言語ではなく，実用システムの構築にも使用しうるプログラミング言語であることを示したものと言える．また，このコンパイラは，ML やその他の関数型言語の新たな機能の実装実験などを行うプラットフォームとしても価値があるものである．

6. ML の研究動向

ML 言語に関する研究はプログラミング言語理論の分野で現在最も活発に研究されているテーマの一つであり，毎年数多くの論文が発表されている．その研究内容も多岐に渡り，これまでに発表された研究成果を網羅的に解説することは不可能である．そこで本章では，ML 言語の研究の基礎となった理論の研究を紹介した後，最近の研究の一部を紹介する．ここで取り上げたトピックは，著者の興味を反映したものになっていることをご了承願いたい．ML の研究動向に興味のある読者は，最近のプログラミング言語に関する代表的な国際会議，たとえば ACM Symposium on Principles of Programming Languages や ACM Conference on Lisp and Functional Programming などの会議録を参照されたい．また，近年 ACM SIGPLAN が ML に関するワークショップを毎年開催している．しかしこれまでは，その会議録は公には出版されていない．

6.1 ML の基礎的性質の研究

ML の理論の基礎となった論文は Milner の多相型型推論理論である[28)]．この論文で Milner は ML の核言語の主要な型判定を推論するアルゴリズム $\mathcal{W}$ を与え，さらに，核言語の指示的意味論を定義し，型推論アルゴリズムがその意味論に関して健全であることを証明した．Damas と Milner[9)]は，ML の核言語の型システムを自然演繹システ

ムとして提示し，Milnerのアルゴリズムがこのシステムに関して完全であることを示した．以上の二つの論文がMLの基礎を与えた論文である．Milnerの論文で定義されたアルゴリズム$\mathcal{W}$は，ML以外の多くのシステムにも採用されている．DamasとMilnerによる型システムの定式化は，簡潔でかつ論理学の形式にしたがった扱いやすいものであり，MLの型システムの形式的な定義の基礎となったばかりでなく，多相型言語の基礎的性質を研究する基礎としても使用されている．

Milnerの論文[28]では再帰的な型は扱われてなかった．MacQueen, Plotkin, Sethiらは Milnerの意味論を再帰的な多相型に拡張し，Milnerによって示された型システムの健全性の定理が，拡張された言語にも成立することを示した[25]．

MacQueenは，**2.**で紹介した彼の提案したStandard MLのモジュールシステム[23]を型理論的に基礎付ける理論を提案した[25]．MitchellとHarperは，モジュールを含んだStandard ML全体を統一的に説明するための理論を提案した[15),33]．

以上の各理論は，伝統的なラムダ式で表現可能な純粋な関数型言語を前提に構築されているため，MLに含まれている非関数的な機能には必ずしも当てはまらない．たとえば，MLの多相型システムの安全性は，非関数的機能を追加すると成り立たないことがすでにMilnerの論文で指摘されていた．現実のMLでは，非関数的な機能の使用にアドホックな制約を加えることによってこの問題を回避していた．Tofteはこの問題を分析し，非関数的な機能をも説明する意味論を提案した[47]．また，Tofteとは独立に，LeroyとWeis[22]やHoang, Mitchell, Viswanathan[18]らも同様の理論を提案した．

6.2 MLの拡張の研究

以上の研究で構築されたMLの理論を一般化し，MLの型システムを拡張しようとする研究が盛んに行われている．

ここ数年特に注目を集めたトピックスの一つに，MLにレコード型やサブタイプの考え方を導入する研究がある．**4.**で紹介したDamasとMilnerの多相型理論は，レコードやバリアントの演算の多相性を表現することができなかった．これらデータ構造は，種々のデータ処理システムにとって必須のデータ構造であり，MLの型システムのこの制限は，MLをデータ処理システムの構築に使用する上での大きな制約であった．Wand[50),51]はこの問題に最初に取り組み，MLの型推論システムをレコード演算に拡張しようと試みた．以降，いくつかの提案がなされ[20),36),42),43),45]またその効率的なコンパイル理論がつくられ[35]，現在ではほぼ解決の見通しがついたと言える．

MLの型システムの拡張の研究としてはそのほかに，並行計算との統合[5),44]，分散処理機能の導入[38]，継続計算(continuation)の導入[10]，より柔軟なモジュールシステムの提案[16),48]，オブジェクト指向の概念との統合[37),52]，外部データ操作のためのダイナミック型の導入[21]，その他多数の研究が現在でも盛んに行われている．

7. おわりに

MLは，堅牢な理論的な基礎に基づき定義された洗練された高水準プログラミング言語である．高品質のコンパイラの出現によって，実用プログラミング言語として普及しつつある．MLはまた，その理論的な厳密さによって，プログラミング言語の理論研究の基礎ともなっている．プログ

処理系名	特徴	問合せ先
Standard ML of New Jersey	Full SML システム	dbm@resarch.att.com
Poly/ML	Full SML システム	ahl@ahl.co.uk
PoplogML	Full SML システム	pop@cs.umass.edu
sml2c	Full SML のC言語へのコンパイラ	dtarditi@cs.cmu.edu
Edinburgh ML	核言語のみ	(ftp アドレス) ftp.dcs.ed.ac.uk
ANU ML	核言語のみ	mcn@anucsd.anu.edu.au
MicroML	核言語のみ	mahler @cs.umu.se
Caml Light	核言語+α	xavier.leroy@inria.fr

図-4 MLの種々の処理系

ラミング言語の新しい機能や既存の機能の分析・拡張の研究がMLを基礎として盛んに行われている．本稿では，MLの以上の二つの側面を紹介した．最後に，MLに関する教科書およびML処理系を紹介する．

MLでのプログラミングの資料としてはHarperのノート[14]がある．入手しづらいかもしれないが，良く書かれたMLの入門書である．近年になって，MLプログラミングの教科書が相次いで出版されている[34),41),46),53]．筆者が読む機会を得た教科書の中では，Paulsonによる文献41)が推薦できる．MLの基礎理論の解説を含んだ教科書には文献54)がある．さらに型理論一般の入門には文献4)，32)，55)などの解説論文が参考になる．

現在のところ最も完成度の品質の高いStandard MLの処理系は**5.**で紹介したStandard ML of New JerseyとAbstract Hardware Ltd.のPoly/MLであろう．Standard ML of New JerseyはAT&Tが著作権を保有するが，ソースコードを含めたシステム全体が無償で配布されており，だれでも自由に使用や改造などを行うことができる．その他種々のシステムが実装されている．**図-4**に最近MLのニュースグループ(comp.lang.ml)にポストされた処理系の情報を紹介する．

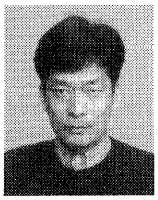
謝辞　本稿の草稿を丁寧に読んで種々の誤りをご指摘下さった沖電気工業(株)の川北泰弘氏，および閲読者に深謝いたします．

参考文献

1) Appel, A. W. and MacQueen, D. B.: A Standard ML Compiler, *Proc. Functional Programming Languages and Computer Architecture* (ed. Kahn, G.), pp. 301-324, Springer-Verlag (1987).
2) Appel, A. W. and MacQueen, D. B.: Standard ML of New Jersey, *Proc. Symposium on Programming Language Implementation and Logic Programming* (ed. Wirsing, M.), Springer-Verlag (1991).
3) Augustsson, L.: Compiler for Lazy ML, *Proc. ACM Symposium on LISP and Functional Programming*, pp. 218-227 (1984).
4) Barendregt, H.: Lambda Calculus with Types, In *Handbook of Logic in Computer Science*, Vol. 2, Oxford University Press (1992).
5) Berry, D., Milner, R. and Turner, D.: A Semantics for ML Concurrency Primitives, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 119-129 (1992).
6) Burstall, MacQueen and Sannella: HOPE: An Experimental Applicative Language, *Proc. ACM Conference on Lisp and Functional Programming* (1980).
7) Cardelli, L.: ML under Unix, *Polymorphism*, **1**, 3 (1983).
8) Constable, R. L. *et al.*: *Implementing Mathematics with Nuprl Proof Development System*, Prentice-Hall (1988).
9) Damas, L. and Milner, R.: Principal Type-Schemes for Functional Programs, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 207-212 (1982).
10) Duba, B., Harper, R. and MacQueen, D. B.: Typing First-Class Continuations in ML, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 163-173 (1991).
11) Gordon, M. J. C.: HOL: A Proof Generating System for Higher-Order Logic, *VLSI Specification, Verification and Synthesis* (eds. Birtwistle, G. and Subrahmanyam, P. A.), pp. 73-128, Kluwer Academic Publishing (1988).
12) Gordon, M., Milner, R. and Wadsworth, C.: *Edinburgh LCF: A Mechanized Logic of Computation, Lecture Note in Computer Science, Springer-Verlag* (1979).
13) Harper, R.: Standard ML Input/Output, *Polymorphism*, **2**, 2 (1985).
14) Harper, R., MacQueen, D. B. and Milner, R.: Standard ML, LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh (1986).
15) Harper, R. and Mitchell, J. C.: On the Type Structure of Standard ML, *ACM Transactions on Programming Languages and Systems*, **15**, 2, pp. 211-252 (1993).
16) Harper, R., Mitchell, J. C. and Moggi, E.: Higher-Order Modules and the Phase Distinction, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 341-354 (1990).
17) Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic, *Trans. American Mathematical Society*, **146**, pp. 29-60 (Dec. 1969).
18) Hoang, M., Mitchell, J. and Viswanathan, R.: Standard ML Weak Polymorphism and Imperative Constructs, *Proc. IEEE Symposium on Logic in Computer Science* (1993).
19) Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel. J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Perterson, J.: Report on Programming Language Haskel, a Non-Strict, Purely Functional Language Version 1.2, *SIGPLAN Notices, Haskel Special Issue*, **27**, 5 (1992).
20) Jategaonkar, L. A. and Mitchell, J.: ML with Extended Pattern Matching and Subtypes *Proc. ACM Conference on LISP and Functional*
21) Leroy, X. and Mauny, M.: Dynamics in ML,

Proc. ACM Conference on Functional Programming Languages and Computer Architecture (1991).
22) Leroy, X. and Weis, P.: Polymorphic Type Inference and Assignment, *Proc. ACM Symposium on Principles of Programming Languages* (1991).
23) MacQueen, D.: Modules for Standard ML, *Proc. ACM Conference on LISP and Functional Programming*, pp. 198-207 (1984).
24) MacQueen, D.: Modules for Standard ML, *Polymorphism*, **2**, 2 (1985).
25) MacQueen, D.: Using Dependent Types to Express Modular Structure, *Proc. ACM Symposium on Principles of Pragramming Languages*, pp. 227-286 (1986).
26) MacQueen, D. B.: Structures and Parametarization in a Typed Functional Language, *Proc. Symposium on Functional Programming Languages and Computer Architechture* (1981).
27) MacQueen, D., Plotkin, G. and Seti, R.: An Ideal Model for Recursive Polymoprphic Types, *Proc. ACM Symposium on Principles of Programming Languages* (1984).
28) Milner, R.:A Theory of Type Polymorphism in Programming, *J. Comput. Syst. Sci.*, **17**, pp. 348-375 (1978).
29) Milner, R.: A Proposal for Standard ML, *Polymorphism*, **1**, 3 (1983).
30) Milner, R.: The Standard ML Core Language, *Polymorphism*, **2**, 2 (1985).
31) Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press (1990).
32) Mitchell, J.: Type Systems for Programming Languages, In *Handbook of Theoretical Computer Science* (ed. van Leeuwen, J.), MIT Press/Elsevier, chapter 8, pp. 365-458 (1990).
33) Mitchell, J. C. and Harper, R.: The Essence of ML, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 28-46 (1988).
34) Myers, C., Clack, C. and Poon, E.: *Programming with Standard ML*, Prentice Hall (1993).
35) Ohori, A.: A Compilation Method for ML-style Polymorphic Record Calculi, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 154-165 (1992).
36) Ohori, A. and Buneman, P.: Type Inference in a Database Programming Language, *Proc. ACM Conference on LISP and Functional Programming*, pp. 174-183 (1988).
37) Ohori, A. and Buneman, P.: Static Type Inference for Parametric Classes, *Proc. ACM OOPSLA Conference* (1989), Extended Version to Apper in *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design* (eds. Gunter, C. and Mitchell, J. C.), MIT Press.
38) Ohori, A. and Kato, K.: Semantics for Communication Primitives in a Polymorphic Language, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 99-112 (1993).
39) Paulson, L. C.: *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge University Press (1987).
40) Paulson, L. C.: Isabelle: The Next 700 Theorem Prover, *Logic and Computer Science* (ed. Odifreddi, P.), pp. 361-386, Academic Press (1990).
41) Paulson, L.C.: *ML for the Working Programmer*, Cambridge University Press (1991).
42) Remy, D.: Typechecking Records and Variants in a Natural Extension of ML, *Proc. ACM Symposium on Principles of Pragramming Languages*, pp. 242-249 (1989).
43) Remy, D.: Typing Record Concatenation for Free, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 166-176 (1992).
44) Reppy, J. H.: CML: A Higher-Order Concurrent Language, *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 294-305 (1991).
45) Stansifer, R.: Type Inference with Subtypes, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 88-97 (1988).
46) Stansifer, R.: *ML Primer*, Prentice Hall (1992).
47) Tofte, M.: *Operational Semantics and Polymorphic Type Inference*, PhD Thesis, Department of Computer Science, University of Edinburgh (1988).
48) Tofte, M.: Principal Signatures for Higher-Order Program Modules, *Proc. ACM Symposium on Principles of Pragramming Languages*, pp. 189-199 (1992).
49) Turner, D.: Miranda: A Non-Strict Functional Language with Polymorphic Types, *Proc. Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, 201, pp. 1-16, Springer-Verlag (1985).
50) Wand, M.: Complete Type Inference for Simple Objects, *Proc. IEEE Symposium on Logic in Computer Science*, pp. 37-44 (1987).
51) Wand, M.: Corrigendum: Complete Type Inference for Simple Object, *Proc. IEEE Symposium on Logic in Computer Science* (1988).
52) Wand, M.: Type Inference for Records Concatenation and Simple Objects, *Proc. IEEE Symposium on Logic in Computer Science* (1989).
53) Wikstrom, A.: *Functional Programming Using Standard ML*, Prentice Hall (1987).
54) 米澤，柴山：モデルと表現，岩波講座ソフトウェア科学，第17巻，岩波書店 (1992).
55) 龍田：型理論Ⅰ～Ⅳ，コンピュータソフトウェア，**8**, 1, 2, 3, 4, pp. 25-33, 40-46, 3-8, 56-68 (1991).

(平成5年5月18日受付)



大堀　淳（正会員）

1957 年生．1981 年東京大学文学部哲学科卒業．同年沖電気工業(株)入社．1989 年米国ペンシルバニア大学計算機・情報科学科博士課程修了．Ph. D.. 1989 年 10 月より 1 年間英国王立協会よりリサーチフェローシップを受け，グラスゴー大学にて客員研究．1990 年 10 月より沖電気工業(株)関西総合研究所に勤務．1993 年 8 月より京都大学数理解析研究所助教授．プログラミング言語の型理論，データモデル理論，オブジェクト指向プログラミングなどに興味をもつ．