■6 群（基礎理論とハードウェア）- 3 編（アルゴリズムとデータ構造）

2 章 データ型とデータ構造

（執筆者：児玉靖司・滝本宗宏）[2012 年 7 月 受領]

■概要■

プログラムには，そのプログラムのもとになっているアルゴリズムによって，より効率的に動作したり，より記憶領域を節約したりできるデータの表現形式が存在する．このような表現形式を**データ構造（data structure）**と呼ぶ．また，例え同じビット列やデータ構造をもつデータであったとしても，そのデータがどのような扱いを仮定しているかによって，処理の仕方が異なる．このようなデータの扱い方を規定する形式を，**データ型（data type）**と呼ぶ．プログラム中の値は，その値に想定されるデータ型と異なったデータ型として処理されると，重大な実行エラーを生ずる可能性がある．このようなエラーを避けるために，多くのプログラミング言語は，値や，値を格納する変数に対して，データ型を指定し，動的に，あるいは静的に，使用されているデータ型の正しさを検査する方法を備えている．

プログラムの実装者は，使用するプログラミング言語で最初から定義されているデータ型を組み合わせて，そのプログラムに都合のよいデータ構造を実現することができる．また，実装者がデータ型を自分で定義できるプログラミング言語では，必要なデータ構造に，そのデータ構造に対して許される操作を規定することによって，新しいデータ型を用意することができる．

このように，データ構造とデータ型は密接な関係があるが，一方で，データ型を，内部で扱うデータ構造と独立に，そのデータ型に対して許される操作の仕様の集合として考える**抽象データ型（abstract data type, ADT）**として定義することができる．この抽象データ型の考え方は，プログラムの効率に直接関係するデータ構造を，プログラム全体の振る舞いから切り離して考慮することを可能にした．現在，抽象データ型の考えは，オブジェクト指向プログラミングというプログラミングパラダイムへと発展をとげ，ソフトウェア開発において，重要な道具となっている．

【本章の構成】

本章ではプログラミング言語に必要なデータ型と，プログラミングの際によく用いられるデータ構造について概観する．2-1 節で，データ型とは何かについて議論し，2-2 節で，データ型の正しさを検証する形式的な方法について述べる．次に，2-3 節と 2-4 節で，静的及び動的なデータ構造を紹介し，これらのデータ構造が，形式的にどのように表現できるかを述べる．最後に各データ構造を記述するプログラミング言語と型について議論する．

 © 電子情報通信学会 2012

■6 群 - 3 編 - 2 章

2-1 データ型

(執筆者：児玉靖司・滝本宗宏) [2012 年 7 月 受領]

概要で述べたように，データの扱い方を規定するデータ型[1]は，そのデータ型をもつ値の処理方法を規定する．例えば，ある 8 ビットのビット列「0x2a」は，整数型とみなして印字処理を行えば「42」が印字され，文字型とみなして処理を行えば「*」が印字される．試しに，以下の C 言語のプログラムを実行してみると，

```
main() {
  char x = 0x2a;

  printf("%d != %c\n", x, x);
}
```

次のように，同じデータが異なって印字されるのが分かる．

```
42 != *
```

もし，0x2a を文字型であると指定し，同じデータ型に対する印字処理だけが許されるという一貫性が保証されるなら，間違って 42 が印字されるような誤りは生じなくなる．このような，データ型に関する誤りのことを一般に**型エラー（type error）**と呼ぶ．

このようなデータ型の一貫性をプログラミング言語に保証させる試みは，プログラミング言語の創生期から行われてきた．データ型の一貫性を検査する方法は，実行時に行う**動的型検査（dynamic type checking）**と実行前に行う**静的型検査（static type checking）**がある．

また，静的型検査で，実行時に型エラーを生じないことを保証するプログラミング言語は，**型付きが強い（strong typing）**といわれる．これに対し，必ずしもデータ型の一貫性を保証しないものは，**型付きが弱い（weak typing）**といわれる[1]．以降で，プログラム中に指定された型（あるいは推論された型）と，値のデータ型について議論するために，プログラムは，**型付きが強いプログラミング言語（strongly typed programming languages）**で記述されているものと仮定する．

プログラミング言語における型の扱いは，上で説明したような素朴な基本型の扱いに関する誤りの検出からはじまり，より複雑なデータ構造に対する型の割当てと型検査，更には抽象データ型やオブジェクト指向など，文脈によって型がもつ情報を区分したり，型どうしに関連をもたせたうえで型検査を行う機能にまで発達してきている．本章ではこれらについて整理して紹介していく．

本章の 2-2 節では，型理論についてその概要を開設している．データ型の「型」という言葉は，20 世紀初頭に数学において「計算の可能性」について議論した**型理論（type theory）**[1]が由来であり，型理論を応用してデータ型の理論的背景を議論することができる．現実のプログラミング言語は必ずしも型理論に基づいて記述可能な型だけを使用しているわけではないが，多くの型付きの強い言語において，**型検査（type checking）**の基盤となるのは型理論による型の割当て可能性であることも確かである．

引き続く 2-3 節と 2-4 節ではそれぞれ，静的データ構造と動的データ構造について取り上

げている．静的データ構造は，プログラムの記述時に構造が定まるようなデータ構造であり，プログラミング言語においてデータを構造化する「土台」となるものである．これに対し，動的データ構造は**ポインタ（pointer）**ないし**参照値（reference value）**を用いて複数のデータ構造を結び合わせて構築されるので，そのかたちはプログラムを実行してみるまで決まらない．なお，ポインタないし参照値の実態は「メモリ上の番地の値」であるが，大部分のプログラミング言語ではこのことは言語仕様上には現れないように注意が払われている．

最後の 2-5 節では，そこまでの内容を組み合わせて，プログラミング言語において型がどのように扱われているかを整理する．例えば，組のような静的なデータ構造にとどまらず，リストや木などの動的なデータ構造であっても，そこに型理論の規則を導入することで型の割当てができることを示している．ただし，一部の値については常に決まった型というかたちでの割当ては行えず，多相型の考え方が必要になる．更に，抽象データ型の場合は，抽象データ型の内部で（実装のために）扱える型の情報と，抽象データ型の外部で（内部表現を参照せずに）型を利用する際の情報を区分する必要があることと，オブジェクト指向では型の間に継承関係があるため，そのことを理論的にも扱う必要がある点を紹介する．

■参考文献

1) A.V. エイホ, R. セシィ, J.D. ウルマン, M.S. ラム　著, 原田 賢一　訳, “コンパイラ 原理・技法・ツール第 2 版,” サイエンス社.
2) Benjamin C. Pierce, “Types and Programming Languages,” MIT Press, 2002.

■6 群 - 3 編 - 2 章

2-2 型の理論

(執筆者：児玉靖司・滝本宗宏)[2012 年 7 月 受領]

データ型は，20 世紀初頭に数学において「計算の可能性」について議論した型理論を発展させることによって，その理論的意味づけを考えることができる．データ型（以降，型理論で扱う場合は，単に**型**と呼ぶ）の基本として，定数（リテラル）及び変数にまず型付けをする．定数 c に対する型 τ を**基底型（atomic type）**という．以降で，複雑な構造をもつデータを形式的に扱うために，λ 式を用いることにしよう．1 章 1-2-2 節で紹介した λ 計算は，型を指定することができないので，仮引数に型を指定できるように拡張する．以降，この拡張した λ 計算を**型付き λ 計算（typed λ calculus）**[2] と呼ぶ．

λ 計算では，定数も λ 式で表現し，文法として定数 c を定義しなかったが，型付き λ 計算では，簡単のために項として定数 c を用意し，型 τ の定数を c^τ と書くことにする．型付きラムダ計算では項 t は以下のように定義する．

$$
\begin{array}{lcll}
t & ::= & c^\tau & \text{（定数）} \\
 & | & x & \text{（変数）} \\
 & | & \lambda x : \tau . t & \text{（}\lambda\text{ 抽象）} \\
 & | & t_1\ t_2 & \text{（適用）}
\end{array}
$$

更に，型 τ は以下のように定義する．

$$
\begin{array}{lcll}
\tau & ::= & a & \text{（基底）} \\
 & | & \tau_1 \to \tau_2 & \text{（関数型）}
\end{array}
$$

関数にも型を定義している点に注意しよう．項 t が型 τ であるとき，$t : \tau$ と書く．また，変数 x と型 τ の関係を集めた集合 Γ を定義し，**型環境（type environment）**[1] または，**型割当（type assignment）**と呼ぶ．Γ は，変数 x_i と型 τ_i を用いて，次のように表すことができる．

$$\Gamma \;=\; \{x_1 : \tau_1,\ x_2 : \tau_2,\ \cdots,\ x_n : \tau_n\}$$

Γ は定義域 $dom(\Gamma) = \{x_1, x_2, \cdots, x_n\}$ から値域 $\{\tau_1, \tau_2, \cdots, \tau_n\}$ への関数と考えることができ，Γ の定義域に変数 x_i が存在するとき，$\Gamma(x_i) = \tau_i\ (1 \leq i \leq n)$ と書く．更に，Γ を用いて以下のように型判定を定義する．

$$\Gamma \;\vdash\; t : \tau$$

Γ のもとで，項 t が型 τ であれば上記の判定は成立する．すなわち，t の自由変数 $FV(t)$ を x とすると，Γ は，x に対する型割当を含んでいなければならない（$FV(t) \subseteq dom(\Gamma)$）．

Γ の要素に，$x : \tau_1$ を追加した集合を $\Gamma\{x : \tau_1\}$ と書くことにすると，項 t の文法に沿った型判定の推論規則は，次のようになる．

$$\Gamma \vdash c^{\tau} : \tau \qquad \text{（定数）}$$
$$\Gamma \vdash x : \tau \quad (\Gamma(x) = \tau) \qquad \text{（変数）}$$

$$\frac{\Gamma\{x : \tau_1\} \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2} \qquad \text{（}\lambda\text{ 抽象）}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2} \qquad \text{（適用）}$$

λ 抽象と適用の規則は,「横線の上を仮定すると，下が言える」と読む．このような推論規則で定義される体系を**型システム（type system）**[1)]と呼ぶ.

それでは，1.2.2 節で紹介したチャーチ数を，推論規則を適用して型判定してみよう．基底型 A を用いて，チャーチ数の 0 は，$c_0 = \lambda s : A \to A.\ \lambda z : A.\ z$ と表せる．この型付き λ 式を型判定すると次のようになる.

1. まず，仮引数 s 及び z の型がそれぞれ $A \to A$，A と指定されているので，式の最も内側の z は，A であると判定できる．形式的に記述すると，次のとおりである.

$$\Gamma\{s : A \to A,\ z : A\} \vdash z : A$$

2. 次に，1. で得られた結果を仮定して，λ 抽象の規則を適用すると次のとおりになる.

$$\Gamma\{s : A \to A\} \vdash \lambda z : A.\ z : A \to A$$

3. 更に，2. で得られた結果を仮定して，λ 抽象の規則を適用すると，最終的に次の型判定結果を得る.

$$\Gamma \vdash \lambda s : A \to A.\ \lambda z : A.\ z : (A \to A) \to (A \to A)$$

以上の推論過程をまとめると，次のとおりである.

$$\cfrac{\cfrac{\Gamma\{s : A \to A,\ z : A\} \vdash z : A}{\Gamma\{s : A \to A\} \vdash \lambda z : A.\ z : A \to A}}{\Gamma \vdash \lambda s : A \to A.\ \lambda z : A.\ z : (A \to A) \to (A \to A)}$$

同様に，チャーチ数 1 の $c_1 = \lambda s : A \to A.\ \lambda z : A \to A.\ s\, z$ についても，適用規則，λ 抽象規則，λ 抽象規則の順に適用することによって，次のように型判定をすることができる.

$$\cfrac{\cfrac{\cfrac{\Gamma\{s : A \to A,\ z : A \to A\} \vdash s : A \to A \qquad \Gamma\{s : A \to A,\ z : A \to A\} \vdash z : A}{\Gamma\{s : A \to A,\ z : A \to A\} \vdash s\, z : A}}{\Gamma\{s : A \to A\} \vdash \lambda z : A \to A.\ s\, z : A \to A}}{\Gamma \vdash \lambda s : A \to A.\ \lambda z : A \to A.\ s\, z : (A \to A) \to (A \to A)}$$

$c_2, c_3, \cdots$ についても同様に型判定することができるので試してみよう．チャーチ数の自然

数について型判定を示したが，演算子や論理値についても同様に型判定ができる．例えば,

$$
\begin{aligned}
plus \quad \equiv \quad & \lambda m :\ (((A \to A) \to (A \to A)).\ \lambda n :\ ((A \to A) \to (A \to A)).\\
& \lambda s :\ A \to A.\ \lambda z :\ A.\ m\ s\ (n\ s\ z)
\end{aligned}
$$

に対して型判定を行うと,

$$
\begin{gathered}
\lambda m :\ (((A \to A) \to (A \to A)).\ \lambda n :\ ((A \to A) \to (A \to A)).\\
\lambda s : A \to A.\ \lambda z :\ A.\ m\ s\ (n\ s\ z) :\\
((A \to A) \to (A \to A)) \to ((A \to A) \to (A \to A)) \to (A \to A) \to (A \to A)
\end{gathered}
$$

という結果を得る．また,

$$
true \equiv \lambda t :\ A_1.\ \lambda f :\ A_2.\ t, \qquad false \equiv \lambda t :\ A_1.\ \lambda f :\ A_2.\ f
$$

については,

$$
true :\ A_1 \to A_2 \to A_1, \qquad false :\ A_1 \to A_2 \to A_2
$$

という結果を得る．

一方，この型システムでは，繰り返し適用を表現する Y コンビネータ[2)]を型判定することができない．

$$
Y \quad = \quad \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))
$$

これは，$(x\ x)$ の部分で，x を x 自身に適用をするような型を割り当てることができないからである．

Y コンビネータの型判定については，多相型を導入することによって，実現できる．多相型については，2.5 節のプログラミング言語と型で触れる．

■参考文献

1)　龍田　真 著, “型理論（レクチャーノート ソフトウェア学）,” 近代科学社.
2)　横内寛文 著, “プログラム意味論（情報数学講座）,” 共立出版.

■6 群 - 3 編 - 2 章

2-3 静的データ構造

（執筆者：児玉靖司・滝本宗宏）［2012 年 7 月 受領］

複数の要素を扱わなければならない問題では，それらの要素をどのようなデータ構造で表現するかが，プログラムの実行効率に大きな影響を及ぼす場合がある．本節では，まず，静的に宣言して扱うデータ構造をいくつか紹介する．これらのデータ構造は，プログラムの記述時に形が定まることから，**静的データ構造（static data structure）**[1)]と呼ばれる，

2-3-1 配　列

配列（array）は，複数の要素を並べた構造をもち，各要素は，**添字（subscript）**を指定することによって参照される．数学では，**ベクタ（vector）**と呼ばれることもある．

ここで，1 つの変数を次のような箱（記憶域）で表現することにしよう．

変数 ：□

一般的な配列は，決まった数の変数を連続したアドレスに割り当てた次のようなデータ構造として表すことができる．

配列 ：

[0]	[1]	[2]	[3]	[4]	[5]

例えば，C 言語では，`int ar[5];` のように宣言し五つの要素をもち添字 [0] ~ [4] として示す配列を宣言することができる．ここで，配列の添字は，0 から始まっているのが分かるが，プログラミング言語によって，1 から始まるものや，添字の下限を定義できるものも存在する．

各要素を参照する際には，次の簡単な式によってアドレスを計算する．

[配列の先頭アドレス] + [要素のサイズ] ∗ [添字]

このような参照の仕方は，すべての要素に同じコスト $O(1)$ でアクセスすることを可能にする．また，一般に，添字の値は静的に決定できないので，すべての要素は，同じデータ型をもつと仮定されることが多い．動的な型検査を行う言語のなかには，任意の型を許すものも存在する．

2-3-2 組

配列は，同じデータ型を要素としてもつことが基本であった．一方，異なった複数のデータ型をもつ値をまとめて扱うためには，**組（tuple）**を用いることができる．

例えば，Standard ML では，`"University"`（文字列），`12.3`（実数），`89`（整数）の組を以下のように簡単に組にすることができる．

```
- ("University",12.3,89); [改行]
val it = ("University",12.3,89) : string * real * int
```

組が生成される際には，同時に，各要素へのアクセス手段が用意される．Standard ML で

は，*n* 個の組の各要素に対して，左から順に#1，#2，···，#n というフィールド名が付与され，要素を参照する際には，その要素の位置に対応するフィールド名を用いる．例えば，2 番目の要素は，次のように参照できる．

```
- #2 ("University",12.3,89);
val it = 12.3 : real
```

2-3-3 レコード

住所録を管理する場合，氏名，電話番号，郵便番号，住所，メールアドレスなどをひとまとめにして扱うことができると都合がよい．このように，異なった値を一つのデータとして扱う仕組みが**レコード（record）**である．レコードのことを**構造体（structure）**と呼ぶこともある．複数のデータをまとめて扱うという意味では，組とよく似ているが，レコードでは，各要素に対してフィールド名を定義できる点が異なる．例えば，氏名，郵便番号，住所に対応するフィールド名,「name」,「post」,「address」をもつレコードを考えよう．Standard ML では，次のように，フィールド名に続いて値を指定することで，レコードを生成することができる．

```
- {name="taro",post="1130022",address="Bunkyo,Tokyo"};[改行]
val it = {address="Bunkyo,Tokyo",name="taro",post="1130022"} :
                    {address:string, name:string, post:string}
```

いったん，レコードが宣言されると，そのレコード値の要素は，対応するフィールド名を使って取り出すことができる．

```
- #address {name="taro",post="1130022",address="Bunkyo,Tokyo"};[改行]
val it = "Bunkyo,Tokyo" : string
```

■参考文献

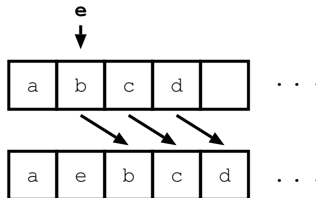
1) 石畑　清 著, “アルゴリズムとデータ構造（岩波講座 ソフトウェア科学 3）,” 岩波書店.

■6 群 - 3 編 - 2 章

2-4 動的データ構造

(執筆者：児玉靖司・滝本宗宏)[2012 年 7 月 受領]

静的データ構造で紹介した配列は，各要素を効率的に参照できるという意味で優れているが，要素を途中に挿入したり，要素を削除した後も要素が連続しているように保ったりするのには，向いていない．例えば，要素 a，b，c，d をもつ配列があるとする．この配列の要素 a の後ろに e を挿入しようとすると，次の図のように，a の後ろに続く要素を一つずつ右にずらさなければならない．もし，決まったサイズの配列が，すべて要素で埋まっている場合には，この操作によって，最後の要素が失われるかもしれない．

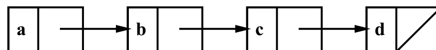


逆に，途中の要素を一つ削除すると，その結果空いた場所を埋めて要素が連続するようにするためには，削除した後に続く要素を一つずつ左にずらさなければならない．

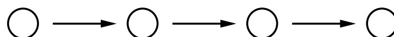
このような要素の挿入や削除を効率的に行うことができるデータ構造として，**動的データ構造（dynamic data structure）**[1)]がある．多くの動的データ構造は，動的に割り当てたメモリ領域をポインタ変数による参照関係で繋いだ構造をもつので，実行中に構造を変化させることが容易である．

2-4-1 リスト

動的に要素を変更することができるデータ構造として最も単純なものが，**リスト（list）**である．リストのデータ構造は，参照関係を矢印で表すと，次のようになる．



リストは，**セル（cell）**と呼ばれる要素とほかのセルへの参照の対（図中の矩形）で構成される．要素は，複数存在してもよい．各セルは，順に次のセルを参照し，最後のセルだけが参照をもたない．リストは次のようにグラフとして表現することもできる．



例えば，Standard ML で，リストを生成するためには，次のようにする．

```
- [“a", “b", “c"]; [改行]
val it = ["a","b","c"] : string list
- [1,2,3][改行]
```

 © 電子情報通信学会 2012

```
val it = [1,2,3] : int list
```

また，次のように，`::`演算子や`@`演算子を用いて，それぞれ，要素を加えたり，二つのリストを繋げたりすることが容易にできる．この演算子（具体的な記号や演算名は言語によって異なる）は，最初に広まったリスト処理言語である **Lisp** における名称をとって **cons** 演算子と呼ぶことが多い，

```
- "d" :: ["a", "b", "c"];
val it = ["d","a","b","c"] : string list
- ["a", "b", "c"] @ ["d", "e"];
val it = ["a","b","c","d","e"] : string list
- ["a", "b"] @ ("c" :: ["d", "e"]);
val it = ["a","b","c","d","e"] : string list
```

一方，リストは，先頭のセルへの参照として扱われ，各セルは，次のセルへの参照しかもたないので，各要素を得るには，先頭のセルから順に参照をたどらなければならない．例えば，Standard ML で，リストの 3 番目の要素を取り出すためには，セルの参照をたどる関数 `tl` とセルの要素を取り出す関数 hd を用いて次のようにする．

```
- hd (tl (tl ["a", "b", "c"])); [改行]
val it = "c" : string
```

一般にリストでは，その要素数が n のとき，要素のアクセスに要するコストは $O(n)$ である．

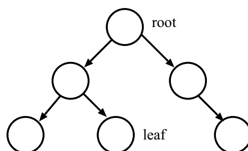
2-4-2 木

節点（node）が無向辺で繋がったグラフ構造のうち，循環が存在しない構造を**木（tree）**という[1)]．辺は必要に応じて有向辺として扱ってもよい．

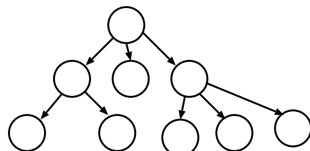
木は，階層構造を表しているので，節点の上下の位置関係を用いた表現が使われる．ある節点のすぐ上にくる節点は，**親**（parent）と呼ばれ，更に辺でたどれる上の節点は，**祖先（ancestor）**と呼ばれる．一方，節点のすぐ下にくる節点は，**子供（child）**と呼ばれ，辺でたどれる下の節点は，**子孫（descendant）**と呼ばれる．また，木の一番上の親の居ない節点は，**根（root）**といい，一番下の子供をもたない節点は，**葉（leaf）**という．

木は，リスト構造において各セルの参照を複数にすることによって実現できる．その際，セルが節点であり，参照が辺に対応する．

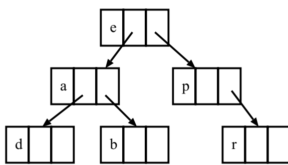
木は，次の (a) のように各節点から最大 2 辺が出ている **2 分木（binary tree）**と，(b) のように 3 辺以上が出ている**多分木（multiway tree）**に分類される．



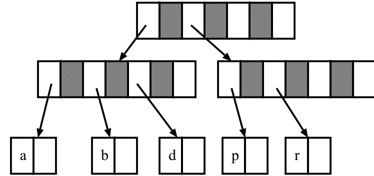
(a) 2 分木


(b) 多分木

データ構造としての木が，いずれの形状をとるかは，そのデータ構造を扱うアルゴリズムによって決まる．データ構造としての 2 分木及び多分木の例には，それぞれ，次の (c) のような **2 分探索木（binary search tree）**や，(d) のような **B 木（B-tree）**がある[1)]．詳しくは，3-3 節で紹介するので，ここでは，構造の概略だけを示そう．



(c) 2 分探索木



(d) B 木

2 分探索木は，節点内に要素 x を格納し，左の子供とその子孫には x より小さい要素，右の子供とその子孫には x より大きい要素を格納するようにした 2 分木である．ある目的の要素をもつ節点を知りたければ，2 分探索木を根からたどり，各節点の要素が，目的の要素より小さければ左の子供，大きければ右の子供というようにたどっていくと，最悪でも木の高さ分の節点を訪問すれば，目的の要素に到達できる．

2 分探索木が有効に働くには，木の左右のバランスが取れていて，その高さが，総節点数より小さい必要がある．左右のバランスが取れるように条件を付加した木は，**平衡木（balanced tree）**と呼ばれる．例えば，各節点の左右の副木の高さの差が，1 以内である **AVL 木（AVL tree）**（3-3 節を参照）も平衡木の一つである．平衡木に要素を付加したり，削除したりする場合は，木全体の再構成が必要になる．

一方，B 木は，多分木の性質を利用して，木のバランスを取る．B 木は，各節点が m 個以下の子供をもつ m **分木**であり，辺と辺の間には，要素の境界値（上記 (b) の節点にある網掛け部分）が与えられている．ある目的の要素をもつ節点を知りたい場合，要素の値が左右の境界値の間に入っている辺をたどっていくことによって，目的の要素を見つけることができる．B 木は，子供が多くなると，節点を分割することによってバランスを取る．

■参考文献

1) 石畑　清, “アルゴリズムとデータ構造（岩波講座 ソフトウェア科学 3）,” 岩波書店.

■6 群 - 3 編 - 2 章

2-5 プログラミング言語と型

（執筆者：児玉靖司・滝本宗宏）[2012 年 7 月 受領]

データ型は，データの扱い方を規定する．データ構造も，データ型を与えることによって，その扱い方という側面から，理解することができる．本章では，2-2 節の型の理論で用いた，型付きラムダ計算を拡張して，これまで紹介したデータ構造のデータ型を考えてみよう．

2-5-1 組　型

組型に対する型判定は，項の文法として項 t_1，t_2 の組 (t_1, t_2) を追加し，その型 $\tau_1 \times \tau_2$ を以下のように定義する．

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1,\ t_2) : \tau_1 \times \tau_2}$$

項 t_1 が τ_1 型で，t_2 が τ_2 型と仮定すると，組 (t_1, t_2) は，$\tau_1 \times \tau_2$ 型をもつ．組型が定義できると，組の i 番目の要素を取り出す関数の型は，次のように決まる．

$$\tau_1 \times \tau_2 \times \cdots \times \tau_i \times \cdots \times \tau_n \ \rightarrow\ \tau_i$$

2-5-2 リスト型

リスト型に対する型判定は，項の文法として項 t_1，t_2 を要素としてもつリスト $[t_1, t_2]$ を追加し，その型 $\tau\ list$ を以下のように定義する．

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash [t_1,\ t_2] : \tau\ list}$$

項 t_1，t_2 が τ 型だと仮定すると，項 $[t_1, t_2]$ は，$\tau\ list$ 型をもつ．リスト型が定義できると，リストの先頭要素を取り出す関数 hd と，2 番目以降のリストを取り出す関数 tl の型は，次のように決まる．

$hd\ : \tau\ list\ \rightarrow\ \tau$

$tl\ : \tau\ list\ \rightarrow\ \tau\ list$

ここで，空リスト [ ] の型は，すべての型の要素に対するリスト型として定義することができるので $\forall\tau.\, \tau\ list$ と書く．このように，「すべての」型を表す全称記号 ∀ を用いた型を**多相型（polymorphic type）**という．多相型についてはすぐ後で説明する。

2-5-3 木　型

木型に対する型判定は，項の文法として項 t_1，t_2，t_3 を要素としてもつ木 $Node(t_1, t_2, t_3)$ と，空の木 $Null$ を追加し，$Node(t_1, t_2, t_3)$ の型 $\tau\ list$ を以下のように定義する．

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau\ tree \qquad \Gamma \vdash t_3 : \tau\ tree}{\Gamma \vdash Node(t_1, t_2, t_3) : \tau\ tree}$$

ここで，$Null$ は，リスト型の空リストと同様に，すべての型に対する空の木として定義すれ

 © 電子情報通信学会　2012

ばよいので，型 $\forall\tau.\,\tau\ tree$ をもつと定義できる．

木型が定義できると，木の根の要素を取り出す関数 $fact$，左の副木を取り出す関数 $left$，及び，右の副木を取り出す関数 $right$ の各型は，次のように決まる．

$$fact\ :\tau\ tree\ \rightarrow\ \tau$$

$$left\ :\tau\ tree\ \rightarrow\ \tau\ tree$$

$$right\ :\tau\ tree\ \rightarrow\ \tau\ tree$$

Standard ML では，配列やリストは言語仕様として備わっているが，木のデータ型は自分で用意しなければならない．木型を定義してみよう．

```
datatype 'a tree = Node of 'a * 'a tree * 'a tree | Null;
exception no_such_value;

fun fact (Node (x,_,_)) = x
|   fact  Null = raise no_such_value;

fun left (Node (_,l,_)) = l
|   left  Null = raise no_such_value;

fun right (Node (_,_,r)) = r
|   right  Null = raise no_such_value;
```

`datatype 'a tree` は，任意の要素型`'a` をもつ `tree` 型の宣言を示している．`Node` と `Null` は，`tree` 型の値を構成する際に，頭に付ける**値構成子（value constractor）**である．`Node` の後には，`'a` × `'a tree` × `'a tree` 型の組が続くことが分かる．

各関数 `fact`，`left`，`right` は，仮引数をパターンとして記述することによって，`Node` の各要素を取り出している．実引数として `Null` が渡されたときには，例外を生ずるようにしている．以下に実行例を示す．

```
- fact;
val it = fn : 'a tree -> 'a
- left;
val it = fn : 'a tree -> 'a tree
- right;
val it = fn : 'a tree -> 'a tree
- fact (Node("a",(Node("b",Null,Null)),(Node("c",Null,Null))));
val it = "a" : string
- left (Node("a",(Node("b",Null,Null)),(Node("c",Null,Null))));
val it = Node ("b",Null,Null) : string tree
- right (Node("a",(Node("b",Null,Null)),(Node("c",Null,Null))));
val it = Node ("c",Null,Null) : string tree
```

2-5-4 多相型

2-2 節で紹介した型付き λ 計算では，Y コンビネータを型判定できないことを述べた．この節では，型付き λ 計算に，次の let 式を加えることによって，Y コンビネータの型判定を可能にする．まず，let 式で拡張した型付き λ 計算を次のように定義する（以下では，プログラミング言語の言い方にならって，項のことを式と呼ぶ）．

$$
\begin{array}{rcll}
e & ::= & c_\tau & \text{（定数）} \\
 & | & x & \text{（変数）} \\
 & | & \lambda x.\, e & \text{（λ 抽象）} \\
 & | & e_1\, e_2 & \text{（適用）} \\
 & | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} & \text{（let）}
\end{array}
$$

(let) 式は，x を e_1 で束縛し，e_2 で使用することを意味するので，$(\lambda x.\, e_1)\, e_2$ と同じ意味を表している．次に，2.2 節の型表現を多相型で拡張する．

$$
\begin{array}{rcll}
\tau & ::= & a & \text{（基底型）} \\
 & | & \tau_1 \to \tau_2 & \text{（関数型）} \\
\sigma & ::= & \tau & \text{（型）} \\
 & | & \forall t.\, \tau & \text{（多相型）}
\end{array}
$$

多相型 $\forall t.\, \tau$ は，型 τ に含まれる自由変数 t が，任意の型であってよいことを表す型である．例えば，$\lambda x.\, x$ は，型 $\forall t.\, t \to t$ となる．更に，型システムを，let 式に対する判定規則で拡張すると，次のようになる．

$$\Gamma \vdash c_\tau : \tau \qquad \text{（定数）}$$

$$\Gamma \vdash x : \tau \qquad (\Gamma(x) \succ \tau) \qquad \text{（変数）}$$

$$\frac{\Gamma\{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2} \qquad (FV(e) \subseteq dom(\Gamma)) \qquad \text{（}\lambda\text{ 抽象）}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \qquad \text{（適用）}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma\,\{x : Cls_\Gamma\,(\tau_1)\} \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau} \qquad \text{（let）}$$

（定数），（変数），（λ 抽象），（適用）は，型付き λ 計算と同じである．（変数）の規則に示した条件 $\Gamma(x) \succ \tau$ は，変数 x が多相型である場合，一貫性のある具体的な型 τ に置き換えることを表している．例えば，$\forall t.\, t \to t$ の具体的な型の一つは $int \to int$ である．$Cls_\Gamma(\tau_1)$ は，τ_1 の自由変数になっている型変数 t を，$\forall t.\tau_1$ のように，任意の型を許すようにすることを意味する．

Y コンビネータで問題となった，「自分自身への適用」は，let $\cdots$ in $\cdots$ end 式を用いることによって，型判定が可能になる．以下は，自分自身への適用」を含む単純な例を示している．

$$
\dfrac{\dfrac{\Gamma\{x : t\} \vdash x : t}{\Gamma \vdash \lambda x.\, x : t \to t} \quad \dfrac{\dfrac{\begin{array}{c}\Gamma\{f : \forall t.\, t \to t\} \vdash f : (int \to int) \to int \to int \\ \Gamma\{f : \forall t.\, t \to t\} \vdash f : int \to int\end{array}}{\Gamma\{f : \forall t.\, t \to t\} \vdash f\ f : int \to int} \quad \Gamma \vdash 3 : int}{\Gamma\{f : \forall t.\, t \to t\} \vdash (f\ f)\ 3 : int}}{\Gamma \vdash \mathtt{let}\ f = \lambda x.\, x\ \mathtt{in}\ (f\ f)\ 3\ \mathtt{end} : int}
$$

let $f = \lambda x.x$ は，f を $\forall t.t \to t$ の型であると判定する．この f の型の型変数 t は，$f\ f$ の変数参照で，一貫性のある具体的な型に置き換えられる．すなわち，$(f\ f)$ の結果が 3 に適用されることから，$(f\ f)$ は，$int \to int$ であり，この結果から，各 f の型は，一つ目が $(int \to int) \to int \to int$，二つ目が $int \to int$ ということになる．

2-5-5 抽象データ型とオブジェクト型

操作対象のデータ構造とは独立に，許される操作の仕様の集合として定義される型を**抽象データ型（abstract data type）**という（詳しくは 4-1 節を参照）．2-5-3 節で，木型を定義したが，木型を表現しているデータ構造を隠蔽することによって，抽象データ型として定義することができる．Standard ML では，abstype を使うことによって，次のようにデータ構造を隠蔽する．

```
abstype 'a tree = Node of 'a * 'a tree * 'a tree | Null with
  exception no_such_value

  fun empty () = Null

  fun makeTree (x,l,r) = Node (x,l,r)

  fun fact (Node (x,_,_)) = x
  |   fact  Null = raise no_such_value

  fun left (Node (_,l,_)) = l
  |   left  Null = raise no_such_value

  fun right (Node (_,_,r)) = r
  |   right  Null = raise no_such_value
end;
```

abstype は，型を定義する点で datatype と同じであるが，型のデータ構造を参照できる有効範囲を，with と end の間に制限する．試しに，abstype の外で，tree 型を扱う関数 fact' を定義してみると，値構成子 Node が未定義になっているのが分かる．

```
- fun fact' (Node (x, _, _)) = x
  |   fact'  Null = raise no_such_value;
stdIn:41.5-42.37 Error: non-constructor applied to argument
 in pattern: Node
stdIn:41.29 Error: unbound variable or constructor: x
```

すなわち，tree 型内に隠蔽されたデータ構造を扱えるのは，abstype 内に定義された関数だけである．abstype の外では，tree 型の値を生成することもできないので，木を生成するための関数 makeTree と空の木を返す関数 empty が加えられている点に注意しよう．これらの関数を実行すると次のようになる．

```
- val t = makeTree("a",makeTree("b",empty(),empty()),
                           makeTree("c",empty(),empty()));
val t = - : string tree
- fact t;
val it = "a" : string
- left t;
val it = - : string tree
- right t;
val it = - : string tree
```

現在，最も多く使われている抽象データ型は，**オブジェクト指向プログラミング（object oriented programming）**におけるオブジェクトであろう（詳しくは 4-2 節を参照）．オブジェクトの型を特に，**オブジェクト型（object type）**と呼ぶことにすると，多くのオブジェクト指向言語がもつ**継承（inheritance）**や，それに類する関係づけによって，**副型（subtype）**と呼ばれる型の概念が必要になる．例えば，型 B のオブジェクト b が型 A のオブジェクト a を継承しているとき，B は A の副型であるといい（逆に，A は B の**上位型**（super-type）という），$B < A$ と書く．このとき，オブジェクト b の型には，次の規則が適用される．

$$\frac{B < A \qquad b : B}{b : A}$$

すなわち，A の副型 B のオブジェクト b は，A 型でもあるということである．この関係から，多くのオブジェクト指向言語では，オブジェクトを上位型の変数や引数に代入することを許している．

例えば，次の Java で記述したプログラムは，main メソッドのなかで，メソッド C1 の A 型の引数として，A を継承した B 型のオブジェクトを渡している．

```
public class A {
  public void A1() {
    System.out.println("calling A1 of A.");
  }
  public void A2() {
    System.out.println("calling A2 of A.");
  }
  public static void main(String args[]) {
    C c = new C();
    c.C1(new B());
  }
}

class B extends A {
  public void A1() {
    System.out.println("calling A1 of B.");
  }
  public void B1() {
    System.out.println("calling B1 of B.");
  }
}

class C {
  public void C1(A a) {
    a.A1();
  }
}
```

この例を実行すると，次のように，クラス *B* でオーバライド（4-2 節参照）されたメソッド *A*1 が呼び出され，A 型の引数として *B* 型のオブジェクトが渡されていることを確認できる．

```
calling A1 of B.
```

一般に，継承やオーバライドだけでなくメソッドの**多重定義**（overloading）が可能なオブジェクト指向言語で書かれたプログラムにおいて，どのメソッドが呼び出されるかは，単純ではない．また，副型の関係は，継承関係と独立に，その型の構造から決まるように一般化することができる．詳しくは，参考文献 1) を参照のこと．

■参考文献

1) Martin Abadi, Luca Cardelli, “A Theory of Objects,” Springer, 1996.

 © 電子情報通信学会 2012