2021年度「プログラミング言語」配布資料 (9)

五十嵐 淳

2022年10月02日

1 多相性

ここまでみてきた2分探索木ではデータとして整数を考えてきたが，実は2分探索木のアイデアは整数データに限られるわけではなく，大小比較ができる (全順序が与えられる) データであれば適用可能である．つまり，アルゴリズムの基本に手を入れることなく浮動小数点数の2分探索木，文字列の2分探索木，さらには整数と文字列のペアの2分探索木などを実装することができる．

例えば Java で，整数のための2分探索木と文字列のための2分探索木は以下のように定義できるだろう．

```
public interface BinarySearchTreeInt {
    boolean find(int n);
    BinarySearchTreeInt insert(int n);
    int min();
    BinarySearchTreeInt delete(int n);
}

public interface BinarySearchTreeString {
    boolean find(String n);
    BinarySearchTreeString insert(String n);
    String min();
    BinarySearchTreeString delete(String n);
}

// Many similar interfaces!?
```

しかし，データの種類毎に，データ型の定義や `find` などの2分探索木操作を与えるのはプログラムを書く観点からも保守する観点からも無駄である．インターフェースに限っていうと，これらの定義で異なっているのは，引数と返値の型の要素型に関する部分だけである．(厳密には `insert` や `delete` の型も異なっているが，これらはインターフェース自身を参照している，という意味では同じようなものと考えられる．)

本章は，このような「型だけが異なる定義が複製される」という問題に対する，プログラミング言語レベルでの解決策を概観する．アイデアは，お馴染みの[1]「異なる部分はパラメータ化してひとつにまとめ，後でパラ

[1]関数定義は，何度も行う類似の計算パターンのうち，異なる式の部分をパラメータ化したものである，と考えると「お馴染み」だろう．例えば，「5 と 3 の和の半分を得る」「100 と 200 の和の半分を得る」という計算の共通部分のみを抽出したのが，「与えられた x と y について，x と y の和の半分を得る」という (`average` という名前の) 関数だ，というわけである．ただし，今回の場合，類似した定義

メータを具体化できるようにする」というものである．詳しい正確な説明は後に譲るが，大雑把には，**要素型をパラメータとした**以下のようなインターフェースを考えよう，ということになる．

```
public interface BinarySearchTree<Elm> { // Elm is a 'type' parameter!
    boolean find(Elm n);
    BinarySearchTree<Elm> insert(Elm n);
    Elm min();
    BinarySearchTree<Elm> delete(Elm n);
}
```

1 行目の <> に挟まれた Elm が型を表すパラメータ (この名前は element から取っている．) で，{} の中であたかも String などのような型と同じように扱うことができる．このような，型でパラメータ化されたインターフェース (やクラス) を**ジェネリック・インターフェース (generic interface)**，**ジェネリック・クラス (generic class)** と呼ぶ．このジェネリック・インターフェース名 BinarySearchTree は Elm にあてはまる具体的な型を (パラメータの宣言と同じく<> を使って) 与えることで，様々なインターフェースとして使うことができる．例えば BinarySearchTree<String> と書くだけで実質的に最初の BinarySearchTreeString (こちらには<> がないことに注意!) と同等のものとして使うことができる．このインターフェース定義での insert や delete の返値型は (1 行目で宣言された)Elm を要素とする 2 分探索木ということで，BinarySearchTree<Elm> が指定されている．(1 行目の BinarySearchTree<Elm> と 3，5 行目の BinarySearchTree<Elm> は字面は全く同じであるが，役割は異なっている．前者はクラスとその型パラメータの宣言，後者はジェネリック・インターフェースへの型引数として Elm が与えられている，ひとつの型を表す表現である．特に，後者の Elm は型パラメータの「使用」である．)

ひとつの定義が型の異なる様々な文脈で使えることを，定義が**多相的 (polymorphic)** である，という．また，多相的である，という性質を**多相性 (polymorphism)** と呼ぶ．多相性にも様々な形態があり，Java における + という演算子が，整数の足し算，浮動小数点数の足し算，文字列の連結に使える，というのも多相性の一種と考えられるが，上記の BinarySearchTree<Elm> のような「定義を型でパラメータ化」することによって発生する多相性を特に**パラメータ多相性 (parametric polymorphism)** と呼ぶ．

以下，パラメータ多相性のための Java と OCaml の言語機構をみていく．C 言語にはパラメータ多相性を直接サポートする機能はないが，ある程度シミュレートすることは可能である．最終的には多相的な 2 分探索木の定義を目指すが，find など，2 分探索木の操作については，さらに準備が必要なことがあるため，まずは，探索のことを考えずに，以下の操作を持つような多相的 2 分木を考える．

- 木のサイズ (これは木の枝の数と定義する) を計算する size
- 木の深さ (根から葉まで到達するために通過する枝の最大数) を計算する depth
- 左右反転させた木を得る reflect
- 木の (一番左の葉の位置に) 新しい要素を追加する add
- 木の文字列表現を得るメソッド toString (Java)，関数 string_of_...tree (OCaml)，もしくは木を表示する関数 (C 言語)

間で異なるのは値ではなく型である．

1.1 多相的 2 分木 in Java (java/polyTree)

既に述べたように，「型のみが異なる定義」をひとつにまとめるために，型情報をパラメータ化したインターフェース・クラス定義を考える．Java では，そういった型情報についてパラメータ化されたものをジェネリック・インターフェース，ジェネリック・クラス (まとめて**ジェネリクス (generics)**) と呼ぶ．また，型情報を表すパラメータを型変数・型パラメータと呼ぶ．型変数はジェネリック・クラス内では，あたかもふつうの (クラスやインターフェースに由来する) 型の名前と同様に，インスタンス変数やメソッドの引数や返値の型として使うことができる．ただし，型変数は「何らかのオブジェクトの型であることがわかっている以外に何もわからない」抽象的な型であるため，型変数に由来する型を持つ式に対して行える操作の種類は非常に限定的である．

1.1.1 2 分木クラスの定義と 2 分木オブジェクトの構成

まず，以下は 2 分木のインターフェース定義である．木に格納されるデータの型は `Elm` という名前の型変数で与えよう．型変数の名前はクラスやインターフェースと同じように選ぶことができる (慣習的に，大文字で始まる名前をつける)．また，簡単のため，まずは `size` と `depth` だけ考えてみよう．

```
public interface Tree<Elm> {
    int size();
    int depth();
    // more signatures to come
}
```

以前であれば，この `Tree` というインターフェースの名前はそのまま型として，インスタンス変数の型やメソッドの引数・返値型として使うことができたが，ジェネリクスの場合，`Elm` が何であるかを指定して使うことになる[2]．例えば，文字列を各 branch に格納した木を表す変数は

```
Tree<String> t = ...;
```

などと宣言することになる．`<>` の中に `Elm` を具体化する **型引数 (type argument)** を書く．`<>` の中には参照型であれば何でも与えることができる．Java では`<>`にプリミティブ型を入れることができず (C# のジェネリクスでは許されている)，以下のような宣言はコンパイラによってエラーとされる．

```
Tree<int> t = ...;  // illegal!
```

その代わりに，Java では各プリミティブ型の値をオブジェクトで表すためのクラス[3]が用意されているので，整数を 2 分木に格納したい場合には，`int` に対応する `Integer` を使って

```
Tree<Integer> t = ...;
```

と宣言する．

次に，葉と枝のクラス宣言をみてみよう．これらのクラスも，要素の型を型パラメータとして宣言して定義することになる．

---

[2] 実は，Java では `Tree` 単体でも型として使えて，誤って`<>` を省略してしまってもエラーも出ない場面も多いのだが，概念的には，`Elm` を必ず指定する，と考えるとよい．

[3] これを (プリミティブ型の) ラッパークラス (wrapper class) と呼ぶこともある．`int` には `Integer`，`double` には `Double` などが用意されている．

```
public class Leaf<Elm> implements Tree<Elm> {
    // no instance variables

    /**
     * Constructor for objects of class Leaf
     */
    public Leaf() {
        // nothing to initialize
    }

    // other methods to define
}

public class Branch<Elm> implements Tree<Elm> {
    // instance variables to hold an element of type Elm and subtrees
    private Tree<Elm> left;
    private Elm v;  // standing for a value
    private Tree<Elm> right;

    /**
     * Constructor for objects of class Branch
     */
    public Branch(Tree<Elm> left, Elm v, Tree<Elm> right) {
        this.left = left;
        this.v = v;
        this.right = right;
    }

    // other methods to define
}
```

既に述べたように，型変数 Elm はクラス定義内であたかもふつうの型であるかのように使える．Branch クラスのインスタンス変数 v の型ではまさにその Elm が型として使われている．また，型変数をジェネリクスの型引数として使うこともできる．その例が implements のところやインスタンス変数の型として出現している Tree<Elm> である．

さて，これらのクラスを使って，実際に 2 分木を構成してみよう．以下が実際に枝や葉のオブジェクトを生成するコードである．変数と同様，オブジェクト生成のキーワード new の後のクラス名も，要素の型を具体的に指定した Branch<Integer> になっている．

```
public class Main {
    public static void main(String[] args) {
        Tree<Integer> t1 = new Branch<Integer>(new Leaf<Integer>(),
                                               Integer.valueOf(10),
                                               new Leaf<Integer>());
```

```
        Tree<Integer> t2 = new Branch<Integer>(new Leaf<Integer>(),
                                               Integer.valueOf(25),
                                               new Leaf<Integer>());
        // Actually, auto boxing allows you to omit "Integer.valueOf()"
        Tree<Integer> t3 = new Branch<Integer>(t1, 15, t2);
        Tree<Integer> t4 = new Branch<Integer>(new Leaf<Integer>(), 60, new Leaf<Integer>());
        Tree<Integer> t5 = new Branch<Integer>(new Leaf<Integer>(), 48, t4);
        Tree<Integer> t6 = new Branch<Integer>(t3, 30, t5);

        // Let's construct a tree holding strings
        Tree<String> t11 = new Branch<String>(new Leaf<String>(),
                                              "I",
                                              new Leaf<String>());
        Tree<String> t12 = new Branch<String>(new Leaf<String>(),
                                              "Love",
                                              new Leaf<String>());
        Tree<String> t13 = new Branch<String>(t11, "Java", t12);
        Tree<String> t14 = new Branch<String>(new Leaf<String>(), "How", new Leaf<String>());
        Tree<String> t15 = new Branch<String>(new Leaf<String>(), "about", t14);
        Tree<String> t16 = new Branch<String>(t13, "you?", t15);

        // to be continued...
    }
}
```

プリミティブ型に対応するクラスのオブジェクトは，丁寧に書くのであれば `Integer.valueOf(10)` と `Integer` クラスの static メソッド `valueOf` を使って生成するが，現在の Java では，`Integer` 型のオブジェクトが期待されている箇所に，単に `10` などと定数 (これはプリミティブ型である `int` 型をもつ) を書くだけで，`Integer.valueOf(10)` のことと解釈してくれる．`t3` 以降はその略記法を使っている．

また，`t11` 以降は，枝に文字列を格納した木を生成している．

1.1.1.1 implements 節と型の代入互換性

クラス定義の `implements` 節は，ふたつの型の間の代入互換性 (どの型の式がどの変数に代入できるか) を与えている．例えば，前章までででみた

```
public class Branch implements BinarySearchTree {
  ...
}
```

というクラス定義であれば，`Branch` オブジェクトを `BinarySearchTree` 型の変数に代入したり，`BinarySearchTree` 型の引数を受け取るメソッドに渡したりすることができる．`implements` 節をそのまま代入互換性の定義として読み替えることができる．

では，`Tree<String>` や `Branch<Integer>` といった型引数を伴った型の間の代入互換性はどうなるのであろうか？ ジェネリクスについては，`implements` 節は，代入互換性を与える一般的な規則として読むことができ

る．例えば

```
public class Branch<Elm> implements Tree<Elm> {
```

であれば「任意の参照型 T について Branch<T> は Tree<T> に代入できる」といった具合である．これによって，上のプログラムでの

```
        Tree<Integer> t3 = new Branch<Integer>(t1, 15, t2);
```

や

```
        Tree<String> t13 = new Branch<String>(t11, "Java", t12);
```

といった代入が正当化できる．また，これ以外の場合 (例えば Branch と Tree に異なる型を与えた場合) には代入互換性がないため，

```
        Tree<String> t13 = new Branch<String>(t1, "Java", t2);
```

などとすると，

```
Main.java:42: エラー: 不適合な型: Tree<Integer>を Tree<String>に変換できません:
        Tree<String> t13 = new Branch<String>(t1, "Java", t12);
                                              ^
```

のように t1 の型 (Tree<Integer>) と new Branch<String> が期待する型 Tree<String> が合致しない，といったコンパイルエラーが発生する．実際，このような代入を許してしまうと，整数の 2 分木と文字列の 2 分木が混ざってしまいまずいことになる．別の言い方をすると，要素の型をその「入れ物」である 2 分木の型の一部とすることで，異なるデータをひとつの 2 分木に混ぜ込んでしまうようなエラーをコンパイラの型検査の時点で検出することができているのである．

この Branch の例では，implements の前後の<> 内に書かれたものが全く同じであるが，一般にはそうではない場合もあるので，代入互換性を導く規則は少し複雑になり，

```
class C<X1,...,Xn> implements I<T1,...,Tm>
```

(ただし T1 … Tm は型変数 X1 から Xn を含みうる型の表現である) という implements 節は，「任意の型 S1…Sn について，C<S1,...,Sn> は I<T'1,...,T'm> (ただし T'i は Ti 中の型変数 X1…Xn をそれぞれ S1…Sn で置換したもの) に代入できる」という規則として読み替えられる．

1.1.2 size と depth

size と depth メソッドは，定義さえ理解すれば，実は特に問題なく書くことができる．

```
    // Leaf クラスに追加
    public int size() { return 0; }

    public int depth() { return 0; }

    // Branch クラスに追加
    public int size() {
```

```
        return left.size() + right.size() + 1;
    }

    public int depth() {
        return Math.max(left.depth(), right.depth()) + 1;
    }
```

これらのメソッドの挙動は，2 分木の形にのみ左右されるもので，格納されたデータ (Elm 型) とは特に関連していないことには注意してもらいたい．だからこそ Elm が何であろうと (どう具体化されようと)，その機能を果たせるのである．この「何であろうと」ということこそが多相性の源泉 (？) である．以下は，size などの呼び出し例である．

```
        // Main クラス main メソッドの続き
        System.out.println("The size of t6 is " + t6.size());
        System.out.println("The depth of t6 is " + t6.depth());

        System.out.println("The size of t16 is " + t16.size());
        System.out.println("The depth of t16 is " + t16.depth());
```

1.1.3 reflect と add

さて，次に reflect と add を考えてみよう．これらのメソッドは新しい 2 分木を返すメソッドとなるので，インターフェースでの返値型は Tree<Elm> となる．これは，Branch クラスで左右の部分木を表す型が Tree<Elm> であったのと同様で，返ってくる 2 分木が格納しているデータの種類がメソッドが起動されたオブジェクトが格納しているそれと同じであることを示している．

```
public interface Tree<Elm> {
    ...
    Tree<Elm> reflect();
    Tree<Elm> add(Elm e);
}
```

また，add は新しい要素を受け取るので，要素を表す型変数 Elm が引数型として使われている．以下，これらのクラス中での定義を示す．

```
    // Leaf クラスに追加
    public Tree<Elm> reflect() { return new Leaf<Elm>(); }

    public Tree<Elm> add(Elm e) {
        return new Branch<Elm>(new Leaf<Elm>(), e, new Leaf<Elm>());
    }

    // Branch クラスに追加
    public Tree<Elm> reflect() {
        return new Branch<Elm>(right.reflect(), v, left.reflect());
    }
```

```
    public Tree<Elm> add(Elm e) {
        return new Branch<Elm>(left.add(e), v, right);
    }
```

ここで，新しい 2 分木を作る際には要素型をクラスの型パラメータを使って指定している．インターフェースでは Tree<Elm> を返すことが要求されているが，インターフェース内では Elm もあたかも通常の型と同様に扱えることを考えると，「任意の型」として Elm を選ぶことで Branch<Elm> オブジェクトは Tree<Elm> が期待されているところに与えることができるようになっている．

また，これらの定義においても，格納されたデータ (Elm 型) と処理内容は特に関連していないことには注意してもらいたい．Elm 型を持つ v こそ登場するものの，メソッドが呼び出された木から新しい木に，右から左に渡されているだけで，v が整数だろうか，文字列だろうが構わないような処理しかしていない．

1.1.4 implements 節について

クラスの implements 節は，クラスがどういうメソッドを定義しなければいけないか，というクラス側の義務を表すが，ジェネリック・インターフェースの場合は，ここで与えた型引数で具体化したメソッドを考えることになる．例えば，クラス Foo の定義が

```
public class Foo implements Tree<String> {
```

と Elm を String で具体化して定義したのであれば，Tree 中の Elm を全て String にして

```
    int size() {...}
    int depth() {...}
    Tree<String> reflect() {...}
    Tree<String> add(String e) {...}
```

を定義する必要がある．ちなみに，この例のように，ジェネリック・インターフェースを実装するクラスがジェネリック・クラスである必要はない．(この場合 Foo オブジェクトは Tree<String> 型の変数に格納することはできるが，その他の Tree<Integer> には格納できない．代入互換性についての一般的な規則の帰結である．)

また，Branch や Leaf のようにジェネリック・クラスにする場合でも，型変数の名前はインターフェースと揃える必要はなく，

```
public class Branch<E> implements Tree<E> {
```

のように書くこともできる．この場合，Branch 本体で使える型名は E なので，インスタンス変数は以下のようになる．

```
    // instance variables to hold an element of type E and subtrees
    private Tree<E> left;
    private E v;  // standing for a value
    private Tree<E> right;

    ...
}
```

また，この場合でも，メソッド定義の義務のルールは同じで「(implements 節の)Tree に与えた型引数 (つまり E) で具体化したようなメソッド」を定義することになり，

```
int size() {...}
int depth() {...}
Tree<E> reflect() {...}
Tree<E> add(E e) {...}
```

という定義になる．

1.1.5 toString による文字列への変換

さて，最後に文字列への変換を行うための toString メソッドを考える．

```
public interface Tree<Elm> {
    ...
    String toString();
}
```

2 分木の文字列表現は，配布資料 (0) に従う．まずは定義をみてみよう．

```
// Leaf クラスに追加
public String toString() {
    return "leaf";
}

// Branch クラスに追加
public String toString() {
    return "branch(" + left + ", " + v + ", " + right + ")";
}
```

Java では，オブジェクトを表す式が，文字列の連結演算子 + の引数のような，文字列を期待する箇所におかれると自動的に toString メソッドを呼び出して文字列に変換する．left と right については，その型は Tree<Elm> なので，今まさに定義しているメソッドを再帰的に呼び出すことになる．一方，v については，その型は抽象的な型パラメータである Elm で，一見，どう文字列に変換していよいのかわからない (別の言い方をすると toString メソッドが呼べるとは限らない) ような気もするが，実は Java では全てのオブジェクトに toString メソッドが備わっており，このような使い方をしても大丈夫である．(よって，自分で Leaf や Branch クラスに toString メソッドを定義せずとも文字列に変換することはできるのだが，どのような文字列に変換されるかはおまかせなので，このように自分で定義できる場合はするのがよい．)

今まで定義したメソッドを使って，左右反転させたり，要素を追加した 2 分木を表示させてみよう．

```
// Main クラスより
System.out.println(t6);
System.out.println("The size of t6 is " + t6.size());
System.out.println("The depth of t6 is " + t6.depth());
```

```
        Tree<Integer> t7 = t6.reflect();
        System.out.println(t7);
        System.out.println("The size of t7 is " + t7.size());
        System.out.println("The depth of t7 is " + t7.depth());

        Tree<Integer> t8 = t6.add(100);
        System.out.println(t8);
        System.out.println("The size of t8 is " + t8.size());
        System.out.println("The depth of t8 is " + t8.depth());
```

実行結果:

```
branch(branch(branch(leaf, 10, leaf), 15, branch(leaf, 25, leaf)), 30, branch(leaf, 48, branch(leaf, 6
The size of t6 is 6
The depth of t6 is 3
branch(branch(branch(leaf, 60, leaf), 48, leaf), 30, branch(branch(leaf, 25, leaf), 15, branch(leaf, 1
The size of t7 is 6
The depth of t7 is 3
branch(branch(branch(branch(leaf, 100, leaf), 10, leaf), 15, branch(leaf, 25, leaf)), 30, branch(leaf,
The size of t8 is 7
The depth of t8 is 4
```

1.2 多相的 2 分木 in OCaml (ocaml/polyTree)

OCaml でも，「型のみが異なる定義」をひとつにまとめるために，Java と同様に型情報をパラメータ化した type 定義を与えることができる．しかし，2 分木を生成したり，2 分木を操作する関数を定義する段になると OCaml と Java には大きな違いが現れる．

1.2.1 2 分木の型定義

OCaml では型変数の名前は (英小文字で始まる) 名前の前に' (アポストロフィ) をつけることになっている．また，型パラメータは型の名前の前に 宣言する．

```
(* Defining the shape of trees *)
type 'elm tree =
    Lf       (* Leaf *)
  | Br of {  (* Branch *)
      left:  'elm tree;
      value: 'elm;
      right: 'elm tree;
    }
```

Java のジェネリクスと同じように，宣言した型変数は = の右側の定義内でふつうの型と同じように使うことができる．ここでは，branch のためのコンストラクタ Br につけられるレコードの要素の型で使っている．

型変数宣言の一般的な構文としては，type ('a, 'b, 'c) foo = ... というように，型変数は括弧内にカンマで区切って並べるが，1 引数の場合は括弧も省略するのが通常である．

この定義を使って，2 分木を作る場合，Java では Leaf や Branch などに型引数を与えて，何を要素とする 2 分木を作ろうとしているかを記述したが，OCaml ではそのような記述は必要なく，単に Lf と Br を使って今までと同様に書けばよい．

```
(* Constructing a sample tree holding integers *)
let t1 = Br {left = Lf; value = 10; right = Lf}
let t2 = Br {left = Lf; value = 25; right = Lf}
let t3 = Br {left = t1; value = 15; right = t2}
let t4 = Br {left = Lf; value = 60; right = Lf}
let t5 = Br {left = Lf; value = 48; right = t4}
let t6 = Br {left = t3; value = 30; right = t5}

(* Now let's construct a tree holding strings *)
let t11 = Br {left = Lf; value = "I"; right = Lf}
let t12 = Br {left = Lf; value = "love"; right = Lf}
let t13 = Br {left = t11; value = "OCaml"; right = t12}
let t14 = Br {left = Lf; value = "How"; right = Lf}
let t15 = Br {left = Lf; value = "about"; right = t14}
let t16 = Br {left = t13; value = "you?"; right = t15}
```

例えば，t1 や t11 の定義をインタラクティブ処理系に与えてみると，

```
# let t1 = Br {left = Lf; value = 10; right = Lf};;
val t1 : int tree = Br {left = Lf; value = 10; right = Lf}
# let t11 = Br {left = Lf; value = "I"; right = Lf};;
val t11 : string tree = Br {left = Lf; value = "I"; right = Lf}
```

と，t1 の型は int tree で，t11 の型は string tree である，と **要素の型を含めて**処理系が推論してくれる．

1.2.2 2 分木操作関数

さて，size などの 2 分木操作関数を定義してみよう．関数定義自体はどれも素直な再帰関数で与えられる．

```
let rec size t =
  match t with
    Lf -> 0
  | Br {left=l; value=v; right=r} -> size l + size r + 1

let max(n, m) =
  if n < m then m else n

let rec depth t =
  match t with
```

```
    Lf -> 0
  | Br {left=l; value=v; right=r} -> max(depth l, depth r) + 1

let rec reflect t =
  match t with
    Lf -> Lf
  | Br {left=l; value=v; right=r} ->
     Br {left = reflect r;
         value = v;
         right = reflect l}

let rec add(t, e) =
  match t with
    Lf -> Br {left=Lf; value=e; right=Lf}
  | Br {left=l; value=v; right=r} ->
     Br {left = add(l, e);
         value = v;
         right = r}
```

ここで注目してもらいたいのは，定義自体ではなく，その型である．例えば，size の型は，インタラクティブ処理系に定義を与えるとわかるが

```
# let rec size t = ...;;
val size : 'a tree -> int = <fun>
```

となる．この，型変数 ('a) が含まれたような型は正確には**型スキーム (type scheme)** と呼ばれ，この関数が多相的に—'a を任意の型で具体化して—使えることを示している．すなわち，size は int tree -> int としても使えるし，string tree -> int としても使える，という具合である．

```
# let s6 = size t6;;
val s6 : int = 6
# let d6 = depth t6;;
val d6 : int = 3
# let s16 = size t16;;
val s16 : int = 6
# let d16 = depth t16;;
val d16 : int = 3
```

型変数 'a は，tree の型定義とは関係なく型推論が勝手に (アルファベット順に) 選んできた名前である．

この多相性は，結局のところ，引数 t が格納している value の型に依存しない処理しか書いていないことに由来するものである．一方，以下のような 2 分木に格納されたデータの和を取る関数 sum

```
# let rec sum t =
    match t with
      Lf -> 0
```

```
    | Br {left=l; value=v; right=r} -> sum l + v + sum r
```

を定義すると，v が整数として使われていることから，その型は

```
val sum : int tree -> int = <fun>
```

と，この関数は整数を格納した 2 分木限定であることを推論してくれる．もちろん sum を文字列を格納した 2 分木に適用することはできない．

```
# sum t11;;
Error: This expression has type string tree
       but an expression was expected of type int tree
       Type string is not compatible with type int
```

このように，OCaml の型推論機能は，引数がどんな使われ方をしているかを調べて，できるだけ多相的な，つまり一般的な，型 (スキーム) を推論してくれるのである．このような，最も一般的な型を**主要型 (principal type)** と呼び，型推論が主要型を推論してくれることを **主要型性 (principal type property)** がある，という．

1.2.3 文字列への変換

Java では文字列の変換を行う際に，全てのオブジェクトには何らかの toString メソッドが定義されていることを利用して型変数 Elm に関わらない一般的な文字列変換処理が記述できた．しかし OCaml にはどんな型でも文字列に変換できるような便利な関数があるわけではないので，要素型に特化した処理をいちいち書くしかない[4]．

```
let rec string_of_int_tree t =
  match t with
    Lf -> "leaf"
  | Br {left=l; value=v; right=r} ->
      "branch(" ^ string_of_int_tree l ^ ","
                ^ string_of_int v ^ ","
                ^ string_of_int_tree r ^ ")"
```

6 行目のライブラリ関数 string_of_int は整数を文字列に変換する int -> string 型の関数なので，先程の sum と同様に，string_of_int_tree の型は int tree -> string となる．

1.3 多相的 2 分木 in C (C/polyTree)

最初に述べたように C 言語には，パラメータ多相を直接サポートする機能はない．このような言語でもある程度シミュレートすることは可能である．そのシミュレートをするアイデアは void * という特殊なポインタ型を使って「何でも格納できる 2 分木」を作る，ということである．void * 型はポインタの型であるが，その使い方は非常に限定されている．他のポインタ型にキャストを使って変換することはできるが，*演算子を使ってそれが指す先の値を取り出すことは実質できない[5]．任意のポインタは，以下のように，キャストを使うこと

[4] Haskell では，この文字列変換のような「処理内容は引数の型に依存するが，多相的に使いたい」関数の定義の負担を軽減する，型クラス (type class) と呼ばれる仕組みが備わっている．

[5] 「実質」と書いているのは，* 演算子を適用すること自体はできるが得られた値の型は，使ってはいけない void 型であるためである．(C 言語仕様 6.3.2.2 参照)

なく (暗黙の型変換を行って)void * 型の変数へ代入することができる.

```
int *i = ...;
void *p = i;   // assignment from int * to void *
```

このことを利用して，多相的データ構造を以下のような方法でシミュレートする.

- データ構造が格納する要素の型を void * に設定する
- データの格納は，データへのポインタを void * として代入する
- データの取り出しは，void * ポインタを，適宜，格納されているはずのデータの型へキャストして使う

具体的にプログラムを見ていこう．まず，型の定義は以下のようになる.

```
struct tree {
  enum nkind { LEAF, BRANCH } tag;
  union {
    struct leaf { int dummy; } lf;
    struct branch {
      struct tree *left;
      void *value;        // a pointer to something
      struct tree *right;
    } br;
  } dat;
};
```

ほとんど，2 分探索木の時と同じだが，branch 構造体の value メンバの型だけが int ではなく void * に変わっている．newbranch, newleaf といった新しい木を作るための補助関数も引数の型を調整する以外は変わらない.

```
struct tree *newbranch(struct tree *left, void *value, struct tree *right) {
  // Allocate a new object in the heap
  struct tree *n = (struct tree *)malloc(sizeof(struct tree));
  // And then initialize the members
  n->tag = BRANCH;            // could be written (*n).tag = BRANCH
  n->dat.br.left = left;
  n->dat.br.value = value;
  n->dat.br.right = right;
  return n;
}

struct tree *newleaf(void) {
  struct tree *n = (struct tree *)malloc(sizeof(struct tree));
  n->tag = LEAF;
  return n;
}
```

これらの定義を使って，整数や (倍精度) 浮動小数点数を格納した 2 分木は例えば以下のように作ることがで

きる.

```
// main 関数より

// Prepare elements in the heap
int *e1 = (int *)malloc(sizeof(int)); *e1 = 10;
int *e2 = (int *)malloc(sizeof(int)); *e2 = 25;
int *e3 = (int *)malloc(sizeof(int)); *e3 = 15;
int *e4 = (int *)malloc(sizeof(int)); *e4 = 60;
int *e5 = (int *)malloc(sizeof(int)); *e5 = 48;
int *e6 = (int *)malloc(sizeof(int)); *e6 = 30;

// Construct a tree
struct tree *t1 = newbranch(newleaf(), e1, newleaf());
struct tree *t2 = newbranch(newleaf(), e2, newleaf());
struct tree *t3 = newbranch(t1, e3, t2);
struct tree *t4 = newbranch(newleaf(), e4, newleaf());
struct tree *t5 = newbranch(newleaf(), e5, t4);
struct tree *t6 = newbranch(t3, e6, t5);

// Now let's construct a tree holding double
double *e11 = (double *)malloc(sizeof(double)); *e11 = 10.1;
double *e12 = (double *)malloc(sizeof(double)); *e12 = 25.2;
double *e13 = (double *)malloc(sizeof(double)); *e13 = 15.3;
double *e14 = (double *)malloc(sizeof(double)); *e14 = 60.4;
double *e15 = (double *)malloc(sizeof(double)); *e15 = 48.5;
double *e16 = (double *)malloc(sizeof(double)); *e16 = 30.6;

// Construct a tree
struct tree *t11 = newbranch(newleaf(), e11, newleaf());
struct tree *t12 = newbranch(newleaf(), e12, newleaf());
struct tree *t13 = newbranch(t11, e13, t12);
struct tree *t14 = newbranch(newleaf(), e14, newleaf());
struct tree *t15 = newbranch(newleaf(), e15, t14);
struct tree *t16 = newbranch(t13, e16, t15);
```

2 分木にはポインタを格納する関係上，まず，データを `malloc` で確保された領域に格納し，それらへのポインタ (変数 `e1` 〜`e6`, `e11` 〜`e16`) を使って木を作っている．`newbranch` の第二引数は定義では `void *` 型であるが，直接 `int *` や `double *` 型の変数を渡すことができている．

どんなポインタでも `void *` の変数に代入できるので，多相データ構造の目的のひとつである「異なる要素型に対してひとつの型定義で対応」することが可能になる．ただし，`int` や `double` のデータを直接構造体に書き込むことはできない．そのためメモリの使用効率は少し (ポインタ 1 つ分) だけよろしくない．

また，2 分木の型は格納しているデータの種類に関わらず `struct tree` になってしまうので，

```
  struct tree *t13 = newbranch(t1, e13, t2);  // t1 and t2 hold (pointers to) integers!!
```

のように，int を格納した 2 分木と，double を格納した 2 分木を (誤って) 繋いだとしても，コンパイラは気付いてくれない．これは Java や OCaml で型レベルで多相性をサポートしていることに比べると，よりプログラマが注意してプログラムを記述しなくてはならないことを意味している．(逆に，わざと，異なる型の値を混ぜたい場合には Java や OCaml では不便な場面もある．Java であれば Object という，全てのオブジェクト型から代入できる型を使って void * と同等なことができる．)

1.3.1 多相的操作の定義

size や depth などの関数は，他の言語の定義からもわかるようにデータ (value メンバ) には手を触れないので，特に変わったところもなく定義できてしまう．

```
int size(struct tree *t) {
  if (t->tag == LEAF) {
    return 0;
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    int s1 = size(b->left);
    int s2 = size(b->right);
    return s1 + s2 + 1;
  }
}

int max(int a, int b) {
  return (a>b)?a:b;   // equivalent to "if a > b then a else b" in OCaml
}

int depth(struct tree *t) {
  if (t->tag == LEAF) {
    return 0;
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    int d1 = depth(b->left);
    int d2 = depth(b->right);
    return max(d1, d2) + 1;
  }
}

struct tree *reflect(struct tree *t) {
  if (t->tag == LEAF) {
    return newleaf();
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
```

```
    struct tree *t1 = reflect(b->right);
    struct tree *t2 = reflect(b->left);
    return newbranch(t1, b->value, t2);
  }
}

struct tree *add(struct tree *t, void *e) {
  if (t->tag == LEAF) {
    return newbranch(newleaf(), e, newleaf());
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    struct tree *t1 = add(b->left, e);
    return newbranch(t1, b->value, b->right);
  }
}
```

異なる種類のデータが混在した 2 分木が作れるのと同じ理由で，`reflect` や `add` が返す 2 分木に格納されたデータの種類が `t` と同じかどうかや，追加する要素 `e` の型が，`t` の要素型と合っているのかなどは型だけ見ていても見えてこない．

1.3.2 2 分木の表示

OCaml と同じく，C 言語には任意のデータを文字列に変換するような機能はないので，2 分木の表示は面倒である．以下は整数を格納した 2 分木の内容を表示する関数である．

```
// Functions whose behavior depends on the element type have to be defined separately
void print_inttree(struct tree *t) {
  if (t->tag == LEAF) {
    printf("leaf");
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    int *v = (int *)b->value;
    printf("branch(");
    print_inttree(b->left);
    printf(", %d, ", *v);
    print_inttree(b->right);
    printf(")");
  }
  return;
}
```

この関数は，`struct branch` の `value` メンバには `int *` が格納されていることを仮定していて，取りだしたポインタはまず，`int *` 型にキャストし (7 行目)，その内容を `int` として表示している (10 行目)．

この関数は，上記の `t1` 〜 `t6` のような `int *` を格納した 2 分木で呼び出されることを想定しているが，もし，

t11 のような double * を格納した 2 分木を渡しても今までと同じ理由でコンパイラは何もエラーを出さないし，元々が double へのポインタだったものを void * に (暗黙) 変換し int * にキャストした場合の動作は未定義となる．(おそらく多くのコンパイラでは何らかの表示はしてくれると思うが．)

1.3.3 おまけ: malloc と free の型

動的にメモリ領域を確保/解放する malloc/free 関数の型をマニュアルで調べてみると，

```
void *malloc(size_t size);
void free(void *ptr);
```

と void * 型のポインタを返す/受け取る関数であることがわかる．これは，このふたつの関数が多相的に使える (どのような型の領域でも扱える) ことを表している，と考えられる．