型レベルプログラミングの会 – 2009/04/18

入門 typeLevel!(D).programming

k.inaba
http://www.kmonos.net/

Dの型レベル計算

- ひとことでいうと

だいたい C++ と同じ

Dの型レベル計算

- 以上。

- ご静聴
  ありがとうございました

まじめに説明

- C++と同じところ
  - template と、
    その 特殊化(specialization) で
    型レベル計算します
- 違うところ
  - 細々と機能が足されてます

今日のおはなしの内容

§１：　D言語の template とは

§２：　型レベル計算のやり方

§３：　応用：“Generic Enumerator もどき”

§４：　応用：“Path-Dependent Type もどき”

基本はだいたいC++と同じなので、
C++と違う部分を中心に・・・

D言語の template のおおざっぱな説明

D言語の template

- C++ の template
  - クラスや関数をパラメタライズする
- D の template
  - “クラスや関数や型定義や変数宣言の集まり” をパラメタライズ
  - Ruby や OCaml の moduleに近い？
  - ※ C++っぽく使うための略記法もあります

例：モジュールっぽい使い方

```
template MyList(T)
{
    class list_t { T car; list_t cdr; }

    list_t nil() { return null; }
    list_t cons( T a, list_t d )
    {
        list_t x = new list_t;
        x.car = a;
        x.cdr = d;
        return x;
    }
}
```

例：モジュールっぽい使い方

```
template MyList(T)
{
    class list_t { T car; list_t cdr; }

    list_t nil() { return null; }
    list_t cons( T a, list_t d )
    {
        list_t x = new list_t;
        x.car = a;
        x.cdr = d;
        return x;
    }
}
```

```
MyList!(int).list_t lst =
    MyList!(int).cons(123,
        MyList!(int).cons(456,
            MyList!(int).nil));
```

例：モジュールっぽい使い方

```
template MyList(T)
{
    class list_t { T car; list_t cdr; }

    list_t nil() { return null; }
    list_t cons( T a, list_t d )
    {
        list_t x = new list_t;
        x.car = a;
        x.cdr = d;
```

```
MyList!(int).list_t lst =
    MyList!(int).cons(123,
                      (456,
                      il));
```

```
mixin MyList!(int);

list_t lst = cons(123, cons(456, nil));
```

別の例：「多相型」の実現

```
template max(T) {
    T max( T x, T y ) {
      if( x < y ) return y;
      else        return x;
    }
}
```

```
int a = max!(int).max(10, 20);
real b = max!(real).max(3.14, 2.72);
string c =
  max!(string).max(“foo”, “bar”);
```

別の例：「多相型」の実現

```
template max(T) {
    T max( T x, T y ) {
      if( x < y ) return
      else          return x
    }
}
```

長すぎる！！！

```
int a = max!(int).max(10, 20);
real b = max!(real).max(3.14, 2.72);
string c =
  max!(string).max(“foo”, “bar”);
```

略記法その1

- 「テンプレートのメンバが1つだけ」
- 「テンプレート名とメンバ名が同じ」なら、

  “. メンバ名” は省略してよい

```
int a = max!(int)(10, 20);
real b = max!(real)(3.14, 2.72);
string c =
  max!(string)(“foo”, “bar”);
```

略記法その2

- 関数テンプレートの場合、
  実際の引数からテンプレート引数が推論できるなら、

  ”!(テンプレート引数)” は省略してよい

```
int a = max(10, 20);
real b = max(3.14, 2.72);
string c = max(“foo”, “bar”);
```

略記法その3

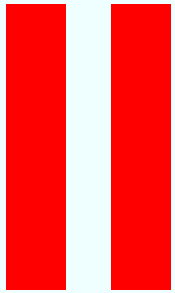
```
template max(T) {
    T max( T x, T y ) {
      if( x < y ) return y;
      else        return x;
    }
}
```

これも
冗長！

- 同名メンバ1つだけを持つ
  関数テンプレートは、
  もっと簡単に書ける

略記法その3

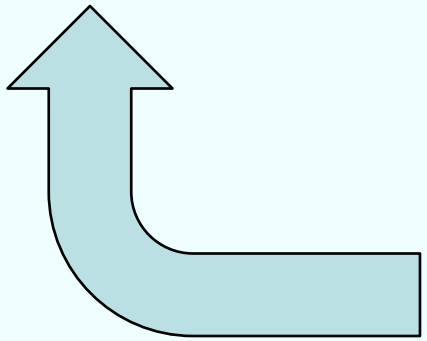
```
template max(T) {
    T max( T x, T y ) {
      if( x < y ) return y;
      else        return x;
    }
}
```



```
T max(T)( T x, T y ) {
  if( x < y ) return y;
  else        return x;
}
```

クラステンプレートでも同様

```
class Stack(T) {
    …実装…
}
Stack!(string) stk = …;
```



```
template Stack(T) {
    class Stack {
        …実装…
    }
}
Stack!(string).Stack stk = …;
```

型レベル計算のやりかた
～aliasと特殊化～

※ちょっと寄り道 alias とは

- 型や変数やテンプレートの別名を定義する

```
alias int 整数; // intの別名”整数”

整数 n = 100;

alias n えぬ; // nの別名”えぬ”
えぬ = 200;   // nに200が入る
```

エイリアステンプレート

- 「1個だけ alias を宣言したテンプレート」を型レベル関数として使うことが多いです。

```
template pointer_to(T) {
  alias T* type;
}

pointer_to!(int).type p; // int*型
```

エイリアステンプレート

- さらに「唯一の同名メンバは “.メンバ名” を省略できる」規則を使うと…

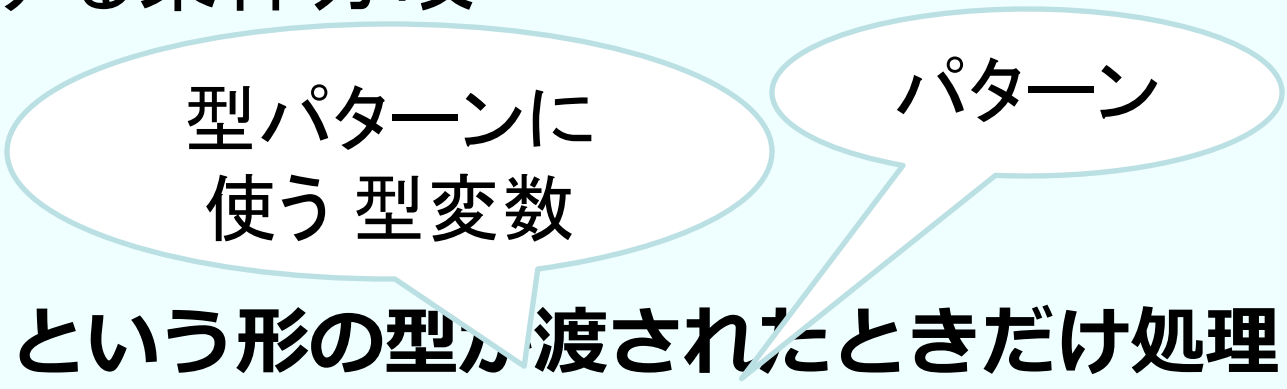
```
template pointer_to(T) {
  alias T* pointer_to;
}

pointer_to!(int) p; // int*型

// 残念ながら↓のような略記はない
// alias T* pointer_to(T);
```

特殊化＝型レベルパターンマッチ

- 型に関する条件分岐




```
// T* という形の型が渡されたときだけ処理
template pointee(T : T*) {
  alias T pointee;
}

pointee!(int*) n; // int型
pointee!(real) r; // コンパイルエラー
```

※ちょっと寄り道

- “Principal Template” (特殊化されてないテンプレート) の宣言が要らない

```
template<typename T>
  struct pointee;        // ←これ
template<typename T>
  struct pointee<T*>
    { typedef T type; };
```

```
template pointee(T : T*)
  { alias T pointee; }
```

定番の例：型リスト

```
class Nil {}
class Cons(A, D) {}

alias Cons!(int,
        Cons!(real,
          Cons!(string, Nil))) MyList;
```

- この型リストの長さを計算する型レベルプログラムを書いてみよう！

定番の例：型リスト

- 解答

```
template length(_: Nil) {
  const length = 0;
}

template length(A: Cons!(A,D), D){
  const length = 1 + length!(D);
}
```

パターン1：
Nilだった時

パターン2：
Consだった時

[これはひどい] 型パターンに
使う2個目以降の変数は
パターンの後ろに記述

static if

- 別解
  もっと手続き型チックに、「static if 文」を使う

```
template len(T) {
  static if( is(T==Nil) )
    const len = 0;
  else
  static if( is(T A==Cons!(A,D),D) )
    const len = 1 + len!(D);
}
```

型レベルプログラミングのための専用構文

- `static if( cond ) {body1} else {body2}`
  - コンパイル時条件分岐
- `is( type comparison )`
  - 型が等しいかどうかの判定
  - 型の種類(クラス?関数?等)判定

型レベルプログラミングのための専用構文

- type func(…)(…) if(条件)

```
T max(T)(T x, T y) if( comparable!(T) )
{
   return x<y ? y : x;
}
```

  - 比較演算ができるような型 T についてのみ、maxを定義
    - concept (C++) や 型クラス(Haskell) のような分割型チェックはない（c.f. boost::enable_if）

型レベルプログラミングのための専用構文

- typeof(式)
  - 式の型を得る

```
typeof(123) n; // nの型はint
```

- is(typeof(式))
  - 式が型エラーのときfalse、OKならtrue

```
T max(T)(T x, T y)
  if( is(typeof(
        valueof!(T) < valueof!(T))) )
{ return x<y ? y : x; }
```

ここまでのまとめ

- C++ と同じく、テンプレートと特殊化（パターンマッチ）で計算を行います
  - `template foo(…){alias … foo;}`
- static if 文や is 式のような、より“普通の” D に近い構文があります
- 「この式で型エラーが起きないなら…」のような変態的な分岐が可能

“タプル”
と
汎用イテレータ

タプルって？

- Dでは、いわゆる「型リスト」が言語のプリミティブとして提供されています。
  - –さっきのはあくまでサンプル

※Dのタプル≒「型（など）のリスト」
あくまでコンパイル時のみの概念。
ML/Haskell/Python等々の”タプル”とは別物

簡単な例

```
class NamedPoint
{
    int x; int y; string name;

    // 引数3つ受け取ってx,y,nameに
    // そのままセットするだけの
    // コンストラクタが欲しい
}
```

汎用性のない解

```
class NamedPoint {
    int x; int y; string name;

    // 超手書き
    this( int x, int y, string n )
      { this.x = x;
        this.y = y;
        this.name  = n; }
}
```

タプルを使ったコピペで使い回せる解

```
class NamedPoint {
    int x; int y; string name;

    this( typeof(this.tupleof) xs )
      { this.tupleof = xs; }
}
```

が引数な
コンストラクタ

の型
(int,int,string)

thisオブジェクトのメンバ変数のリスト
(x,y,name)

※ライブラリ化

```
class NamedPoint {
    int x; int y; string name;
    mixin SimpleConstructor;
}

template SimpleConstructor()
{
    this( typeof(this.tupleof) xs )
      { this.tupleof = xs; }
}
```

タプルは型ではなくて型リスト

- この関数は (A,B,C) と D を受け取る
  2引数関数ではなく、
  ```
  void foo( TypeTuple!(A,B,C), D );
  ```
  A,B,C,D を取る4引数関数

- “T…” でタプルパラメタ
  ```
  void printf(TS…)(string fmt, TS xs);
  ```
  2引数関数ではなく可変個引数関数

応用例："Generic Enumerator"

- データ構造から全自動でイテレータを作る！
  - http://www.kmonos.net/wlog/80.html#_1008071221

    …の簡単バージョン。全要素に関数適用！！

```
class List(T) {
  T         car;
  List!(T) cdr;
}

List!(char) lst = …;
each( lst, (char c){writeln(c);});
```

each を List 専用にすれば簡単

- 再帰的にデータをたどるだけ

```
void each(T,F)( List!(T) lst, F fn )
{
    if( lst !is null ) {
      fn( lst.car );
      each( lst.cdr, fn );
    }
}
```

応用例："Generic Enumerator"

えーマジList専用!?
キモーイ

List専用が許されるのは小学生までだよねー

```
class BinTree(T) {
  T               value;
  BinTree!(T) left;
  BinTree!(T) right;
}

BinTree!(real) bt = …;
each( bt, (real r){ writeln(i); });
```

要は

- 渡されたものが List でも BinTree でも、「渡されたもののフィールド全部」を取ってこれれば良い
- あと、型がわかれば良い
  - T なら関数適用、Foo!(T) なら再帰的に潜る…という条件分岐に型情報が必要

実装

```
void each(T, alias Cont, F)
          (Cont!(T) cont, F fn )
{
 if( cont !is null )
  foreach(field; cont.tupleof)
   static if( is(typeof(field)==T) )
     fn( field );
   else static if
      ( is(typeof(field)==Cont!(T)) )
     each( field, fn );
}
```

こまかい拡張 (1)

- コンテナのクラスが直接再帰すると思ったら大間違いだ！

```
Class Tree(T) {
    class Node {
      T       value
      Node[] children;
    }
    Node root;
}
```

- 「Cont!(T) を見つけたら再帰的に」じゃなくて「メンバ(のメンバ)*にTがあるなら再帰的に」じゃないとまずい

そのように実装する

```
void each_impl(T, X, F)( X x, F fn ){
  static if( exists!(T).inside!(X) )
     static if( is(X == T) )
        fn( x );
     else if( x !is null )
        static if( is(X E == E[]) )
           foreach(elem; x)
              each_impl!(T)(elem,fn);
        else
           foreach(field; x.tupleof)
              each_impl!(T)(field,fn); }
void each(T, alias C, F)(C!(T) c, F fn)
  { each_impl!(T)(c, fn); }
```

型レベル計算
exists!(T).inside!(X)

- Xを再帰的にたどりながら T があるかチェック
  - ただし無限ループしないように、一度たどった型は覚えておく（グラフの深さ優先探索）

```
template exists(T) {
  template inside(X)
    { const inside = rec!(X).value; }

  template rec(X, Visited...) {
    static if( indexOf!(X, Visited)!=-1 )
       const value = false;
    else
       … 再帰的に X.tupleof をたどるコード …
  }}
```

こまかい拡張 (2)

```
class AVLTree(T) {
  int             height;
  AVLTree!(T) left;
  T               value;
  AVLTree!(T) right;
}
AVLTree!(int) set = …;
each( set, (int n){ writeln(n); });
```

要素の型 int が内部データの型と一致する時だけが問題！

AVLTree!(real) でも AVLTree!(string) でも問題ないのに！！

- 「AVLTree!(int) の中の int」を列挙すると height まで表示されてしまう！！！！！

そりゅーしょん！

- AVLTree!(内部データに使われない型) なら問題ないなら、AVLTree!(内部データに使われない型)を使えばいいじゃない

小技：Shadow Type

```
void each_impl(ST, SX, X, F)( X x, F fn ){
  static if( exists!(ST).inside!(SX) )
    static if( is(SX == ST) )
      fn( x );
    else if( x !is null )
      static if( is(SX SE == SE[]) )
        foreach(elem; x)
          each_impl!(ST,SE)(elem,fn);
        else
          foreach(i,field; x.tupleof)
           each_impl!(ST,typeof(SX.tupleof[i]))
                     (field,fn); }
void each(T, alias C, F)(C!(T) x, F fn)
  { class ShadowT {}
    each_impl!(ShadowT,C!(ShadowT))(x, fn); }
```

型レベルでは C!(ShadowT) のなかの ShadowT を探す。実行時は C!(T) の中の T。

※いろいろ細かいＱ＆Ａ

- Shadow Type について
  - Q: コンテナが特殊化されてる場合は？
    （C++ の vector<bool> の中の bool の位置は vector<ST> の中の ST の位置と違う！）
  - A: コンテナを特殊化するな！
  - Q: コンテナがif制約を使ってる場合は？
    （class AVLTree!(T) if(comparable!(T)) {…}）
  - A: class ShadowT { T t; alias t this; } でだいたい突破できます

※いろいろ細かいＱ＆Ａ

- Generic Enumerator 全般
  - Q: データ型のフィールドが private だったら？
  - A: tupleofはprivateを完全無視する超仕様…
  - Q: データ型がオブジェクト指向的に多態で実装されてる場合の扱いは？（ class Tree(T){ interface Node{…}; Node root; … } ）
  - A: ごめん、これは無理！＞＜
  - Q: 双方向リストを渡すと無限ループしない？
  - A: ごめん、これも無理！＞＜

“alias引数”
と
パス依存型もどき

さまざまな引数

- 「型」レベルというかどうか微妙なんですが

- Dのtemplateは、引数に色々とれます
  - 型
  - 値（整数、浮動小数点数、文字列）
  - テンプレート
  - alias to
    - グローバル/ローカル/メンバ変数
    - グローバル/ローカル/メンバ関数、無名関数リテラル

※寄り道：値パラメタ

- フィボナッチとか書けるんですが

```
template fib(int n) {
  static if( n<=1 ) const fib = 1;
  else const fib =
          fib!(n-1)+fib!(n-2);
}

int array[fib!(10)];
// int array[89]; と同じ
```

※寄り道：CTFE

- コンパイル時でも普通の関数を呼べるので、計算だけなら普通はそっちで書きます。

```
int fib(int n) {
  return
    n<=1 ? 1 : fib(n-1)+fib(n-2);
}
int array[fib(10)];
```

- 副作用のない（&オブジェクト等を使わない）関数が、定数が必要な文脈で使われていると定数に畳み込むことを保証

それはともかく alias 引数の話

- alias を引数にとれます

```
void setter(alias x)(typeof(x) p)
  { x = p; }

class Foo {
   private int n;
   alias setter!(n) set_n;
}
```

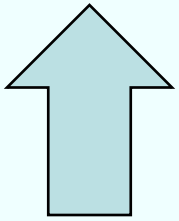
- これを使って遊んでみましょう

突然ですが

- 僕が C++ でよくやるバグ

```
vector<int> v = ≪someprogram≫;
for_each(v.begin(), v.end(), f);

vector<int> u = ≪someprogram≫;
for_each(u.begin(), v.end(), f);
```



※ iterator：凄く適当な説明

- コンテナ中の位置を指すオブジェクト
  - * で、指してる要素をget
  - ++ で、次の要素を指すようにする
  - c.begin() で先頭, c.end() で末尾が取れる

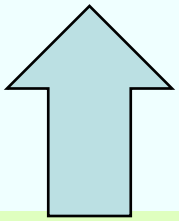
```
void for_each
 (Ite,F)(Ite from, Ite to, F fn) {
   while( from != to ) {
      fn( *from );
      ++from;
   }}
```

再掲：僕が C++ でよくやるバグ

- v.end() と u.end() が
  同じ型なのが悪い

```
vector<int> v = ≪someprogram≫;
for_each(v.begin(), v.end(), f);

vector<int> u = ≪someprogram≫;
for_each(u.begin(), v.end(), f);
```



そりゅーしょん

- alias 引数！

```
class iterator(alias cont)
  { … 実装は略 … }

iterator!(cont)
  begin(alias cont)() {
    return new iterator!(cont)(
      cont.begin()
    );
  }          // end も同様
```

そりゅーしょん

- これならコンパイル時に型エラー

```
vector!(int) v = ≪someprogram≫;
vector!(int) u = ≪someprogram≫;
for_each(begin!(u), end!(v), f);
```

こっちは
iterator!(u) 型

こっちは
iterator!(v) 型

End

- まとめ
  - テンプレートとパターンマッチによる型レベル計算
  - static if や is 式など、型レベル処理専用構文
  - .tupleof のようなコンパイル時リフレクション
  - 型や整数の他に、「変数へのalias」も型レベルで扱える → 何かもっと面白いことができそう…