Функциональный взгляд на визиторы

Visitors Revisited

А. М. Пеленицын
apel@sfedu.ru

Южный федеральный университет
Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра информатики и вычислительного эксперимента

15 апреля 2016
Семинар «Языки программирования и компиляторы»

Содержание

Задача

- Дано: гетерогенная древовидная структура.
- Требуется: применять к каждому узлу набор операций.

Пример реализации чистого ОО-подхода

Определение типов узлов

```
interface Exp {
    int eval();
}
class Lit implements Exp {
    int x;
    Lit(int x)   { this.x = x; }
    public int eval() { return x; }
}
class Add implements Exp {
    Exp l, r;
    Add(Exp l, Exp r)
        { this.l = l; this.r = r; }
    public int eval()
        { return l.eval() + r.eval(); }
}
```

Использование

```
Exp e = new Add(
            new Add(
                new Lit(4),
                new Lit(2)),
            new Lit(1)
        );
out.print(e.eval()); // -> 7
```

Выводы

- *Легко*: добавлять типы узлов.
- *Трудно*: добавлять операции.

Визитор: противоположная ситуация

Определение типов узлов

```
interface Exp {
    void accept(Visitor v);
}
class Lit implements Exp {
    int x;
    Lit(int x)   { this.x = x; }
    public void accept(Visitor v)
        { v.visitLit(this); }
}
class Add implements Exp {
    Exp l, r;
    Add(Exp l, Exp r)
        { this.l = l; this.r = r; }
    public void accept(Visitor v)
        { v.visitAdd(this); }
}
```

Визитор и его использование

```
interface Visitor {
    void visitLit(Lit l);
    void visitAdd(Add a);
}
class EvalVisitor implements Visitor {
    int val;
    public void visitLit(Lit l)
        { val = l.x; }
    public void visitAdd(Add a) {
        a.l.accept(this); int n = val;
        a.r.accept(this); val += n;
    }
}
// ...
EvalVisitor ev = new EvalVisitor();
e.accept(ev);       // e ~ (4 + 2) + 1
out.print(ev.val); // -> 7
```

Пример реализации чистого функционального подхода

Определение типов узлов

```
data Exp
        = Add Exp Exp
        | Lit Int

eval (Add e1 e2) = eval e1 + eval e2
eval (Lit n)     = n
```

Использование

```
let e = Add
            (Add
                (Lit 4)
                (Lit 2))
            (Lit 1)
in print (eval e)       -- -> 7
```

Выводы

- *Легко*: добавлять операции.
- *Трудно*: добавлять типы узлов.

Симуляция функционального решения в ОО-подходе с помощью downcast

Определение типов узлов

```
interface Exp { /* empty */}
class Lit implements Exp {
    int x;
    Lit(int x)   { this.x = x; }
}
class Add implements Exp {
    Exp l, r;
    Add(Exp l, Exp r)
        { this.l = l; this.r = r; }
}
```

Операция

```
static int eval(Exp e) {
    if (e instanceof Lit)
        return ((Lit)e).x;

    else if (e instanceof Add)
        return eval(((Add)e).l) +
               eval(((Add)e).r);

    throw new RuntimeException(
               "Expression type unknown");
}
```

Чем это плохо?

Никаких гарантий согласованности определений операций и типов узлов.

Scala case-classes: AlgDT для бедных

«Видишь downcast? — А он есть!»

```
sealed abstract class Exp
case class Lit(x : Int) extends Exp
case class Add(l : Exp, r : Exp) extends Exp

object Main extends App {

    def eval(e : Exp) : Int = e match {
        case Lit(x)    => x
        case Add(l, r) => eval(l) + eval(r)
    }
    val e = Add(Add(Lit(4),Lit(2)),Lit(1))
    print(eval(e)) // -> 7
}
```

Однако есть проверка полноты (totality checking): уберём `case Lit`

```
warning: match may not be exhaustive.
It would fail on the following input: Lit(_)
def eval(e : Exp) : Int = e match {
                          ^
```

Expression Problem

Джереми Гиббонс, У. Оксфорда, 2013 г.

…This dichotomy is reminiscent of that between OO programs structured around the ***Visitor pattern*** *[9] and those in the* ***traditional OO style*** *with methods attached to subclasses of an abstract* Node *class. The challenge of getting the best of both worlds — extensibility in both dimensions at once — has been called* ***the expression problem*** *[10].*

[10] Филипп Вадлер, Java-genericity mailing list, 1998

1. Расширяемость в обоих измерениях (типы узлов и операции)…
2. …без модификации старого кода или дублирования кода.
3. Статический контроль типов.
4. Раздельные трансляция и контроль типов.
5. (*) Расширение за счёт независимых разработок.

(*) — Zenger и Odersky (2005).

Расширяемое определение типов узлов

Ссылка на статью.

Отдельные типы узлов

```
data Lit e = Lit Int
data Add e = Add e e
```

Сумма типов

```
data (f :+: g) e = Inl (f e)
                 | Inr (g e)
```

Рекурсия

```
data Fix f = In (f (Fix f))

type Expr = Fix (Lit :+: Add)
```

Пример выражения $(4 + 2) + 1$

```
e :: Expr
e = In (Inr (Add
             (In (Inr (Add
                       (In (Inl (Lit 4)))
                       (In (Inl (Lit 2)))))
             )
             (In (Inl (Lit 1)))
    )
)
```

Замечание. Так никто не пишет, надо определять умные конструкторы:

```
e = lit 4 ⊕ lit 2 ⊕ lit 1
```

Функтор: применение операций к узлам

Напоминание

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [a] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap _ [] = []
    fmap f (x:xs) = f x : fmap f xs
```

Типы узлов и суммы — функторы

```
instance Functor Lit where
    fmap f (Lit x) = Lit x

instance Functor Add where
    fmap f (Add e1 e2) = Add (f e1) (f e2)

instance (Functor f , Functor g) =>
    Functor (f :+: g) where
        fmap f (Inl e) = Inl (fmap f e)
        fmap f (Inr e) = Inr (fmap f e)
```

Свёртка по рекурсивному типу

```
foldFix :: Functor f => (f a -> a) -> Fix f -> a

foldFix f (In t) = f (fmap (foldFix f) t)
```

Визиторы и расширяемость

Визитор-вычислитель

```
class Functor f => Eval f where
    evalA :: f Int -> Int

instance Eval Lit where
    evalA (Lit x) = x

instance Eval Add where
    evalA (Add x y) = x + y

instance (Eval f, Eval g) =>
    Eval (f :+: g) where
        evalA (Inl x) = evalAlgebra x
        evalA (Inr y) = evalAlgebra y

eval :: Eval f => Fix f -> Int
eval expr = foldFix evalA expr

-- e :: Fix (Add :+: Lit)
main = print (eval e) -- -> 7
```

Расширяемость узлов

Для добавления нового узла:

- определить АДТ для него,
- экземпляры нужных визиторов (классов),
- опционально: умные к-торы.

Например,

```
data Mul x = Mul x x

instance Functor Mul where
    fmap f (Mul x y) = Mul (f x) (f y)

instance Eval Mul where
    evalA (Mul x y) = x * y

-- e :: Fix (Add :+: Lit :+: Mul)
main = print (eval e)
```

Расширяемость операций (визиторов): создание новых классов.

Визитор печати

```
class Render f where
        render :: Render g => f (Fix g) -> String

pretty :: Render f => Fix f -> String
pretty (In t) = render t

instance Render Lit where
        render (Lit i) = show i

instance Render Add where
        render (Add x y) = "(" ++ pretty x ++ "+" ++ pretty y ++ ")"

instance (Render f , Render g) => Render (f :+: g) where
        render (Inl x) = render x
        render (Inr y) = render y

main = print (pretty e, eval e)
```