Trees

Rob Miller
Fall 2007
Data type patterns

- Tuple type $T_1 \times T_2$
- Option type $Tag_1(T_1) \cup Tag_2(T_2)$
- List-like type $T = Tag_1(...xT) \cup Tag_2(...)$

Iterator

- A state machine for stepping through the elements of a data structure
More Recursive Types

Last time we saw list-like recursive types

- for BigInt: \[ \text{ListRep} = \text{Digit}(\text{int} \times \text{ListRep}) \cup \text{Zero()} \]
- generic list pattern: \[ \text{List}<E> = \text{NonEmpty}(E \times \text{List}<E>) \cup \text{Empty()} \]

Today we’ll look at tree-like recursive types

- Example: symbolic expressions over rationals
  \[ x^2 + 2x + 1 \quad (x-5)^3 \quad (xy)/(yz) + 1 \]

\[ \text{Expr} = \text{Const}(\text{Rat}) \]
\[ \cup \text{Plus}(<\text{Expr} \times \text{Expr}>) \]
\[ \cup \text{Minus}(<\text{Expr} \times \text{Expr}>) \]
\[ \cup \text{Times}(<\text{Expr} \times \text{Expr}>) \]
\[ \cup \text{Divide}(<\text{Expr} \times \text{Expr}>) \]
\[ \cup \text{Pow}(<\text{Expr} \times \text{int}>) \]

multiple occurrences of \text{Expr} in one recursive case is what makes this a tree, rather than a list

© Robert Miller 2007
Implementing Expr in Java

interface Expr {
    Rat eval();
}

class Const implements Expr {
    private Rat r;
    public Const(Rat r) { this.r = r; }
    public Rat eval() { return r; }
}

class Plus implements Expr {
    private Expr e1, e2;
    public Plus(Expr e1, Expr e2) {
        this.e1 = e1; this.e2 = e2;
    }
    public Rat eval() {
        return e1.eval().plus(e2.eval());
    }
}
Convenient Factory Methods

Factory methods are creators for data type values

- Simplifies client code

```java
class Exprs {
    private Exprs() {} // private constructor prevents Exprs from being instantiated, since it’s just a holder for static methods
    public static Expr const(Rat r) {
        return new Const(r);
    }
    public static Expr const(int n) {
        return new Const(new Rat(n));
    }
    public static Expr const(String s) {
        return new Const(new Rat(s));
    }
    public static Expr plus(Expr e1, Expr e2) {
        return new Plus(e1, e2);
    }
    ...
    // minus(), times(), divides(), pow()
}
```

© Robert Miller 2007
Using Expressions

import static Exprs.*;

Expr e = plus(const(1), const("3/2"))
Rat result = e.eval();
Expressions with Variables

Expr = Const(Rat)
∪ Var(String)
∪ Plus(Expr x Expr)
∪ Minus(Expr x Expr)
∪ Times(Expr x Expr)
∪ Divide(Expr x Expr)
∪ Pow(Expr x int)

class Var implements Expr {
    private String name;
    public Var(String name) {
        this.name = name;
    }
    ...
}

class Exprs {
    ...
    public static Expr var(String name) {
        return new Var(name);
    }
}

Expr e = plus(pow(var("x"), 2), var("x"));
// e is x^2 + 2x

Expr f = times(var("x"), var("y"));
Evaluating Expressions with Variables

Evaluation requires a value for the variable

class Var implements Expr {
    public Rat eval() {
        ???
    }
}

Design 1: Pass an environment mapping vars to vals

interface Expr {
    public Rat eval(Map<Var,Rat> environment);
}

Design 2: Substitute for all vars before evaluating

interface Expr {
    Expr e
    public Expr subst(Var v, Rat r);
    public Rat eval();
}

let’s use Design 2

Substituting an Expr for a variable makes subst() more generally useful

© Robert Miller 2007
Designing Exceptions

**eval() may still fail**

- If caller neglects to substitute for some variable, then Var.eval() will be called

```java
class Var implements Expr {
    public Rat eval() {
        throw new NoValueException(name + " has no value");
    }
}
```

**Checked vs. unchecked exceptions**

- Exception (and its subclasses) are **checked**: caller must either catch the exception or declare it in the method’s throws clause
- RuntimeException (and its subclasses) are **unchecked**: compiler doesn’t care if it’s caught or declared
- Rule of thumb:
  - if a correctly-written caller can *always avoid* the exception, then use an unchecked exception. (e.g. NullPtrExc, ArrayIndexOutOfBoundsException)
  - If not, then use a checked exception. (e.g. IOException)
More Operations: Subst()

```java
interface Expr {
    Expr subst(Var v, Expr sub);
}

class Const ...
{
    public Expr subst(Var v, Expr sub) {
        return this;
    }
}

class Var ...
{
    public Expr subst(Var v, Expr sub) {
        if (this.equals(v)) return sub;
        else return this;
    }
}

class Plus ...
{
    public Expr subst(Var v, Expr sub) {
        return plus(e1.subst(v,sub), e2.subst(v,sub));
    }
}
```

© Robert Miller 2007
interface Expr {
    Expr deriv(Var v);
}

class Const {
    public Expr deriv(Var v) {
    }
}

class Var {
    public Expr deriv(Var v) {
    }
}

class Plus {
    public Expr deriv(Var v) {
    }
}
deriv() is Chopped Into Little Pieces

deriv() is a function over the recursive Expr data type

- deriv: Expr x Var → Expr

- Imagine if we could write it that way in Java:

```java
Expr deriv(Expr e, Var v) {
    switch (e) {
        case Const c: return const(0);
        case Var v2: if (v2.equals(v)) return const(1);
                     else return v2;
        case Plus p:  return plus(deriv(p.e1, v), deriv(p.e2, v));
        case Pow p:   return times(const(p.k), times(pow(p.e, p.k-1),
                                         deriv(p.e, v)));
        ...
    }
}
```

Some languages can do this (e.g. ML), but not Java

Implementing deriv() as a method is problematic

- Splits its definition across multiple source files, making it harder to understand and maintain

- Creating new functions (integrate()? simplify()?) will require changing Expr and all its subclasses

© Robert Miller 2007
Visitor Pattern

A visitor object represents a function over a recursive data type

- The visitor “visits” each object in a tree or list, just as a recursive function would

```java
interface Visitor {
    Expr onConst(Const e);
    Expr onVar(Var e);
    Expr onPlus(Plus e);
    Expr onPow(Pow e);
    ...
}

interface Expr {
    ...
    Expr accept(Visitor v);
}

class Const implements Expr {
    ...
    public Expr accept(Visitor v) {
        return v.onConst(this);
    }
}

class Plus implements Expr {
    ...
    public Expr accept(Visitor v) {
        return v.onPlus(this);
    }
}

...

© Robert Miller 2007
deriv() as a Visitor

public static Expr deriv(Expr e, Var v) {
    return e.accept(new Deriv(v));
}

Tree to traverse

Parameters needed during traversal

class Deriv implements Visitor {
    Var v;
    public Deriv (Var v) { this.v = v; }

    public Expr onConst(Const e) {
        return const(0);
    }
    public Expr onVar(Var e) {
        if (e.equals(v)) return const(1);
        else return e;
    }
    public Expr onPlus(Plus e) {
        return plus(e.e1().accept(this), e.e2().accept(this));
    }
    ...
}

Need observer methods that extract the parts of each type, since DerivVisitor may be defined outside of Expr or its package

Traverse the descendents of this Plus node

© Robert Miller 2007
Generalizing Visitor

Functions defined with visitors may need various return types, not just Expr

```java
interface Visitor<T> {
    T onConst(Const e);
    T onVar(Var e);
    T onPlus(Plus e);
    T onPow(Pow e);
    ...
}
```

```java
interface Expr {
    ...
    T accept(Visitor<T> v);
}
```
Exercise: Degree of an Expression

public static int degree(Expr e, Var v) {
    return e.accept(new Degree(v));
}

degree(x^2 + 2x + 1, x) = 2
degree( (x-5)^3 + y^4, x) = 3

class Degree implements Visitor<Integer> {
    private Var v;
    public DegreeVisitor(Var v) { this.v = v; }

    public Integer onConst(Const e) {

    }
    public Integer onVar(Var e) {

    }
    public Integer onPlus(Plus e) {

    }
    public Integer onPow(Pow e) {

    }
    ...
}

© Robert Miller 2007
Visitor for Functions Over Option Types

Recall that functions over option types and recursive types have a particular form

- Function over option types have multiple cases (one per type tag)
- Functions over recursive types are frequently recursive

Visitors support both

Expr = Const(Rat)
  ∪ Var(String)
  ∪ Plus(Expr x Expr)
  ...

```java
class DerivVisitor ...
{
  public Expr onConst(Const e) { ... }
  public Expr onVar(Var e) { ... }
  public Expr onPlus(Plus e) {
    return plus(e.e1().accept(this),
               e.e2().accept(this));
  }
  ...
}
```

multiple cases

recursion

© Robert Miller 2007
Visitor Over Option Type

Visitors are useful for option types without recursion too

- Example: message passing in the CommonSenders example

Message = Senders(String x Set<String>) ∪ Done()

class HandleMessage implements Visitor<Void> {
    ...
    public void onDone(Done msg) {
        commonSenders.doneFolder();
    }
    public void onSenders(Senders msg) {
        commonSenders.gotSomeSenders(msg.folder(), msg.senders());
    }
}
Patterns for Functions Over Options

**Function as methods**
- Has privileged access to representation
- Harder to understand the function, because it’s split over multiple files
- Adding a new function requires changing every option class
- Use for operations that are essential to the abstract data type

**Function as visitor**
- Requires observer operations for getting parts of the rep
- Function is found all in one place, so is easier to understand & change

**Function as switch statement**
- Java doesn’t provide good support for this, or for the equivalent if-then:
  ```java
  Expr deriv(Expr e, Var v) {
    if (e instanceof Const) { Const c = (Const) e; ... }
    else if (e instanceof Var) { Var v2 = (Var) e; ... }
    else if (e instanceof Plus) { Plus p = (Plus) e; ... }
  }
  ```
- Compiler doesn’t check that you’ve covered all the cases!
- In **methods** and **visitor** pattern, compiler checks that every case is handled

© Robert Miller 2007
Data as Computation

**A visitor is an object representing a computation**

- DerivVisitor class computes deriv: Expr x Var → Expr
- A DerivVisitor instance (e.g. new DerivVisitor(var(“x”))) computes d/dx : Expr → Expr
- To call a visitor (i.e. run the computation it represents), pass it to accept()

**Using data to represent computation is a powerful idea**

- It allows us to treat a computation as a **first-class object** that can be passed & returned from methods, stored in variables and other data types, even manipulated

© Robert Miller 2007
We’ve seen another example of data-as-computation in this lecture: Expr itself!

- Compare these two representations for $xy+1$:

```java
Rat f (Rat x, Rat y) {
    return x.times(y).plus(new Rat(1));
}
```

Running the computation

- $f$($new$ Rat(2), new Rat(3))
- $g$.subst(var(“x”),const(2)).subst(var(“y”),const(3)).eval()

Manipulating the computation

- We saw how to compute a derivative of an Expr: deriv($g$, var(“x”))
- But it’s not obvious how to do symbolic differentiation on Java code like $f$
Interpreter Pattern

Expr is an example of the interpreter pattern

- An interpreter is a data type representing the syntax of a language
- In this case, Expr represents the language of algebraic expressions over rational numbers

Interpreters are often tree-like data types

- A particular computation in the language is represented as a tree (often called an abstract syntax tree, or AST)

Interpreters allow manipulating the computation, not just running it

- An AST can be run (e.g. with subst() and eval())
- But it can also be inspected and manipulated (e.g. deriv(), simplify(), degree())

Interpreter makes a language expression first-class
Summary

Tree types
- Recursive types that take the form $T = \text{Tag}_1(... \times T) \cup \text{Tag}_2(...)$

Visitor pattern
- An object that represents a function over an option type or recursive type

Interpreter
- A data type that represents the syntax of a language

Data values that represent computation
- Interpreter and visitor are data values representing a computation that’s ready to unfold