Constructing a Tree using the Prolog

September 30, 2020

0.1 Constructing a Tree using the Prolog

- Review some topics from the course.
- Learn how to reperesent the trees in Prolog.
- Implement some tree related operations in Prolog.

0.1.1 Tuples

One of the simplest ways of building compound data structures is by using the tupes. Here is instances of tuples in Haskell-like programing language:

```
(1,2,3):(Int, Int, Int)
("Hello",1): (String, Int)
("Hello"):(String)
():()
```

What is the difference between lists and tuples?

- Tuples are always heterogeneous, whereas lists are often homogeneous.
- (that is, tuples can contain a mixture of types).
- Tuples have a fixex length (built into the type).

So, how we can present tuples in prolog?

- A Prolog compound term of the form **label(a1,...,an)** can be viewed as an n-ary tuple along with the label *label*, and **we will use this fact to construct trees in Prolog**.

0.2 The original tree types

Lets recall the types **BinTree** and **LeafTree** from homework 1:

```
sealed trait LeafTree[A]
case class Leaf[A](a: A) extends LeafTree[A]
case class Branch[A](l: LeafTree[A], r: LeafTree[A]) extends LeafTree[A]

sealed trait BinTree[A]
case class Empty[A]() extends BinTree[A]
case class Node[A](l: BinTree[A], a: A, r: BinTree[A]) extends BinTree[A]
```

For the reminder of the course, if we discuss these types, we will assume **constructors** of these shape.)

0.3 Tupling the arguments

Consider the parameters of each constructor. - ***Leaf*** has a single parameter of type ***A***. - ***Branch*** has two parameters of type ***LeafTree A***. - ***Empty*** does have a parameter, of type ***Unit***. - The only value of type ***Unit*** being ***()***. - ***Node*** has three parameters of types ***BinTree A***, ***A***, and ***BinTree A***.

Considering this we could isomorphically define constructors which each took a single **tuple** as parameter.

- ***Leaf*** would have a parameter of type ***Tuple1[A]***.
  - To construct a singleton tuple with value ***v***, use ***Tuple1(v)***.
  - For instance, ***Tuple1(5) : Tuple1[Int]***.
- ***Branch*** would have a parameter of type ***Tuple2[LeafTree A, LeafTree A]***.
- ***Empty*** is the same as ***Empty***, taking a parameter of type ***Unit***.
  - There is no ***Tuple0*** type in Scala, but ***Unit*** is ***isomorphic***.
- ***Node*** would have a parameter of type ***Tuple3[BinTree A, A, BinTree A]***.

We have to say ***isomorphically*** rather than ***equivalently*** because these constructors as **not** equivalent to the previous versions (except for **Empty**.) But they are **isomorphic**, because they can represent the same trees, and we have a 1-1 correspondence between them.

The Haskell naming of the tuple type would make these descriptions briefer. - ***Leaf*** would have a parameter of type ***(A)***. - ***Branch*** would have a parameter of type ***(LeafTree A, LeafTree A)***. - ***Empty*** would have a parameter of type ***()***. - ***Node*** would have a parameter of type ***(BinTree A, A, BinTree A)***.

0.4 Trees without constructors

Given the above constructors using tuples, we can see that we could even ***omit*** the constructors and simply write trees ***as tuples***. For instance,

```
Branch(Leaf(1),Branch(Leaf(2),Leaf(3))) : LeafTree[Int]
```

corresponds to the tuple

```
(1,(2,3)) : Tuple2[Int,Tuple2[Int,Int]]
```

They are not the same type, but they represent the same tree.

Of course we may introduce confiusion and junk in the same time. However, introducting of confiusion is not a really big problem, since by having look at the ***shape*** of the tuple we would understand the kind of tree it represent. But we will introduce some junks. ***Because there is no static way to seperate a tuple that represent a tree from the tuples that represent other kind of data***

- In a statically typed language such as Scala and Haskell,this method of representation is practically unusable for this reason.
- But in a *dynamically* typed language (we encourage you to read "dynamically typed" as "dynamically type checked", as Pierce suggests in his chapter 1) where no types are checked until runtime, this approach is feasible, and in the absence of user-defined types, necessary!

0.5 Recognising trees

Recall that a Prolog compound term of the form ***label(a1,…,an)*** can be viewed as an **n**-ary tuple along with the label ***label**. We will use the labels to indicate the constructor we have in mind when constructing trees as tuples.

So, for the ***LeafTree*** type, we have trees such as

```
leaf(5)
leaf([])
branch(leaf(1),branch(leaf(2),leaf(3)))
branch(branch(leaf(1),leaf(2)),leaf(3))
```

and for **BinTree**, examples include

```
empty
node(empty,1,empty)
node(node(empty,'left element',empty),top_element,node(empty,3,empty))
```

Considering these definitions, we can construct predicates to check our two tree "types".

These allow for *runtime* checking that arguments have the "correct type" (Type checkers).

```
isBinTree(empty).
isBinTree(node(L,_,R)) :- isBinTree(L), isBinTree(R).

isLeafTree(leaf(_)).
isLeafTree(branch(L,R)) :- isLeafTree(L), isLeafTree(R).
```

There is nothing said with the above predicates about the data inside the trees. So we can easily have trees mix togeather different kinds of data. But the big question to ask is do we really need to check if the tree has the right shape? and the data in the tree has the same type? As an example we can see the list in prolog.

[1]:
```
%%script swipl -q
append([a],[1],X).
```

```
X = [a, 1].
```

0.6 Operations on Trees

Let's implement some basic operations on our tree type. The **flatten** and **orderedElems** operations from homework 1 will be assigned as **homework**.

```
% Inserting into the empty tree creates a node containing E,
% with empty subtrees.
binInsert(E,
          empty,
          node(empty,E,empty)).
```

To insert into a non-empty `BinTree` (a `node`) we must use a recursive clause.

Naively, we might want to write, for instance, `binInsert(E, node(L,A,R), node(binInsert(E,L),A,R))`.

- Notice how what we intend to be the ***"recursive call"*** is **not** the same predicate; binInsert has **three** arguments, not **two**.
- In any case, a predicate is either true or false; it doesn't return a "value".

So we need a recursive premise instead.

```
% Inserting into a node
% inserts it into the left subtree.
% (This implementation arbitrarily chosen.)
%
binInsert(E,
          node(L,A,R),
          node(NL,A,R)) :- binInsert(E,L,NL). % NL for "New Left"
```

In the above, we made an arbitrary choice about where to insert the new element. Specifically, we inserted it as far left as we could. This is a decent choice, as far as it goes; but note that in a logical language, we don't really have to make a choice! ***We can give as many recursive clauses as we like, and then when a user makes an insert query, they could choose the response (solution) that best fits their need.***

```
% Inserting into BinTrees *nondeterministically*.
% This version could be made to produce all possible valid inserts!

% There's only one way to insert into the empty tree.
binInsertND(E,empty,node(empty,E,empty)).

% But there are at least 2 ways we can insert into a nonempty tree.
binInsertND(E,node(L,A,R),node(NL,A,R)) :- binInsertND(E,L,NL).
binInsertND(E,node(L,A,R),node(L,A,NR)) :- binInsertND(E,R,NR).
```

Lets try this quickly to see if it works?

```
binInsert('hello world!!',empty,X).

binInsert('hello worl!!', X, node(empty, 'hello worl!!', empty)).
```

Now based on what we did for `BinTree`, lets see how we can write the same clauses for `leafTree`:

```
% Inserting into a leaf results in a branch with two leaves.
leafInsert(E,
           leaf(A),
           branch(leaf(A),leaf(E))).
```

```
% Inserting into a branch inserts it into the right subtree.
% (again, this is an arbitrary choice.)
leafInsert(E,
           branch(L,R),
           branch(L,NR)) :- leafInsert(E,R,NR).



% Inserting into LeafTrees nondeterministically.

% We have two choices for inserting into a leaf.
leafInsertND(E,
             leaf(A),
             branch(leaf(A),leaf(E))).
leafInsertND(E,
             leaf(A),
             branch(leaf(E),leaf(A))).

% And there are at least 2 ways we can insert into a branch.
leafInsertND(E,
             branch(L,R),
             branch(L,NR)) :- leafInsertND(E,R,NR).
leafInsertND(E,
             branch(L,R),
             branch(NL,R)) :- leafInsertND(E,L,NL).
```

Lets test this implementations too see if it works at its desired?

```
leafInsert(6, branch(branch(leaf(1),leaf(2)), branch(leaf(3),leaf(4))),X).
```

What if we try the nondeterministic implementation?

```
leafInsertND(6, branch(branch(leaf(1),leaf(2)), branch(leaf(3),leaf(4))),X).
```

Now lets implement the `binDelete` based on the above implementation.

```
binDelete(E, node(empty,E,empty),empty).
binDelete(E, node(L,A,R), node(NL,A,R)) :- binDelete(E, L, NL).
```

Lets test it out:

```
binDelete('h', X, node(empty, 'hi', 'habib')).
binDelete('h', X, node(empty, 'hi', node(empty,'habib',empty))).
binDelete('h', node(node(empty,'h',empty),'hi', node(empty,'habib',empty)), X).
```

Exercise: Implement the nonedeterminstic version of delete for both Bin and Leaf trees.

As practice in tutorial, we worked out a way to "join" two trees together.

```
leafJoin(leaf(E1),
```

```
        leaf(E2),
        branch(leaf(E1),leaf(E2))).
leafJoin(leaf(E1),
        branch(L,R),
        branch(L,branch(leaf(E1),R))).
leafJoin(branch(L,R),
        leaf(E2),
        branch(L,branch(leaf(E2),R))).
leafJoin(branch(L1,R1),
        branch(L2,R2),
        branch(branch(L1,R1),branch(L2,R2))).
```

Now lets see if this is workig?

```
leafJoin(branch(leaf(1),leaf(2)), branch(leaf(3),leaf('hi')),X).
```

Not necessarly correct but arbitary implementation:

```
leafJoin(leaf(E1),
        branch(L,R),
        branch(NL,R)) :- leafJoin(leaf(E1),L,NL)
```

What exactly this implementation doing?

Exercise: Implement the nonedeterminstic version of joint for both Bin and Leaf trees.