Trees in Prolog

Mark Armstrong

September 29, 2020

Contents

1 Introduction

These notes were created for, and in some parts **during**, the lecture on September 25th and the following tutorials.

2 Motivation

So far in this course,

- we have had a homework on defining different tree datatypes in Scala, and
- we have discussed in lectures how trees are used as the internal and formal representation of programs.

Now, in keeping with this theme, let us discuss how we can reason about trees in Prolog. This information will be necessary for the sort of programs we wish to write later on.

3 Datatypes in Prolog

We have previously discussed that there are four classes of Prolog terms;

- *numbers*,
  - (including both integers and floats)
- *atoms*, which include words beginning with lower case letters and strings in single quotes,
- *variables*, which begin with upper case letters, and
- *compound terms*
  - (which consist of a *functor* atom and a number of *arguments* applied to that functor,
  - such as `isPrime(5)`.)

These are, in fact, the basic types defined in the Prolog standard.

- See the SWI Prolog documentation.
- There is some hierarchy among the types (see the next section.)
- Other types, or even the ability for user-defined types, may be added as extensions. But in basic Prolog, there are only the above.

Of course, we use in SWI Prolog the list type it provides as well. But for this course, that is the only extension we will use. So we must find a way to represent trees with the above.

4 Aside: the type hierarchy in Prolog

In the above linked SWI Prolog documentation on types, David Tonhofer in the comments links to his Prolog type chart and notes on the subject of SWI Prolog types.

We include that type chart here for your interest.

SWI Prolog 8.1 data type tree

```
                                                                     any T
                                                                       |
                                            +--------------------------+------------------------+
                                            |                                                   |
                                          var(T)                                             nonvar(T)
                                   ("is T a variable name                                       |
                                    that is still fresh                                         |
                               variable at this point in time?")                                |
                                                                    +---------------------------+-----------------------------+
                                                                    |                                                         |
                                                                atomic(T)                                                 compound(T)
                                                                    |                                                         |
                                +-----------------------------------+------------------------+                 +--------------+-------------+
                                |                                   |                        |                 |                            |
                            blob(T,_)                            string(T)                number(T)       "compound term            "compound term
                                |                                                            |             of arity 0"                of arity > 0"
                                |                                                            |                                              |
        +-----------------------+------------------------+                         +---------+---------+                   +----------------+-------------+
        |                       |                        |                         |                   |                   |                |             |
 (other blob types)    blob(T,reserved_symbol)      blob(T,text)           rational(T,Nu,De)        float(T)              dict          "head of       ...others
encapsulated foreign            |                     atom(T)                      |                                       |            of a list"
     resources                  |                        |                         |                                (this seems to     '[|]'(H,Rs)
                                |                        |                         |                              be an encapsulated      [H|Rs]
                       +--------+--------+               |              +----------+----------+                     data structure)         |
                       |                 |               |              |                     |                                       (a nonlocal
                     T==[]             T\==[]            |   rational(X),\+integer(X)      integer(T)                               structure; there
                  empty list        dict functor         |      (proper rational)                                                    may or may not
                                                         |                                                                            be an actual
                                                         |                                                                        list beyond the head)
                                      +------------------+------------------+
                                      |                  |                  |
                                lenghth=0            length=1            atom with
                             "the empty atom"       "character"          length>1
```

The detailed SVG can be quite interesting to examine; it is far too dense to fit on this page, though.

5 Tuples

From Pierce's "Types and Programming Languages", (chapter 11, "Simple Extensions")

> Most programming languages provide a variety of ways of building compound data structures. The simplest of these is pairs, or more generally tuples, of values.

For instance, in Haskell-like notation, the following are tuples.

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

Tuple types differ from lists in that

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

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.

6 Trees as tuples

6.1 The original tree types

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]
```

or in briefer Haskell syntax,

```
data LeafTree a = Leaf a | Branch (LeafTree a) (LeafTree a)
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
```

(for the remainder of the course, if we discuss these types, we will assume constructors of this shape.)

6.2 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`.

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)`.

6.3 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, this tuple representation would introduce a lot of *junk*;

- the set of all tuples

contains many tuples which are not part of

- the set of all tuples which represent a well-formed `LeafTree` (or `BinTree`.)

Also note that with the tuple representation, the type of the tree depends upon how many elements are in it! 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!

7 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))
```

We can construct predicates to check our two tree "types". These allow for *runtime* checking that arguments have the "correct type".

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

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

Note that we still have nothing restricting the types of the elements.

8 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. (Updated September 26th: the original versions of `binInsert` and `binInsertND` did not actually produce trees. They needed the recursive calculation to be a premise.)

```
% Inserting into trees.

% 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))`. But notice how what we intend to be the "recursive call" is not the same predicate; `binInsert` has three arguments, not two. And 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).
```

Everything is similar for `LeafTree`'s. We make some different arbitrary choices about where to insert here, just because we can.

```
% 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).
```

As practice in tutorial, we worked out a way to “join” two trees together. Write alternative ways if you like!

```
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))).
```