Tree Isomorphism: An Exercise in Functional Programming

Jayadev Misra

9/10/01

1 Problem Description

The problem is to decide if two unordered trees are the same. More precisely, define a binary relation *isomorphic* over non-empty trees by the following rules.

1. A tree with a single node (the root) is isomorphic only to a tree with a single node.
2. Two trees with roots A and B, none of which is a single-node tree, are isomorphic iff there is a 1-1 correspondence between the subtrees of A and of B such that the corresponding subtrees are isomorphic.

Shown below are two trees that are isomorphic; corresponding nodes are labeled with the same label.

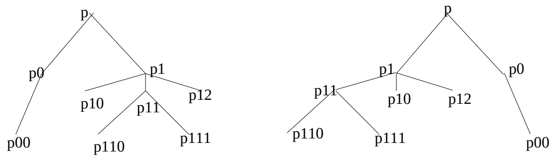

Figure 1: Isomorphic Trees

I describe an algorithm in Haskell to decide if two given trees are isomorphic.

2 An Algorithm

Bag-Representation of a Tree Associate the following representation with trees. A tree with a single node is represented by the empty bag. A tree that has subtrees $T_0, \ldots, T_N$ is represented by $\{t_0, \ldots, t_N\}$, where t_i is the representation of T_i, $0 \le i \le N$. Two trees are isomorphic provided the bags representing them are equal. Our algorithm for isomorphism can be used as the basis for deciding bag equality (and hence, set equality).

List-Representation of a Tree Define type `Utree` (unordered tree) by

```
data Utree = Tree[Utree]
```

That is, a `Utree` is constructed from a list of `Utree`s by applying the constructor `Tree`. There is no circularity in this definition; the ground term is `Tree[]`, which denotes a tree with a single node. Since this definition avoids case analysis —distinguishing between singleton and non-singleton trees— the algorithm for isomorphism also avoids case analysis.

Example: The first tree given in the problem description is defined by the following declarations.

```
leaf = Tree[]
p110 = leaf
p111 = leaf
p00 = leaf
p10 = leaf
p12 = leaf
p11 = Tree[p110,p111]
p0 = Tree[p00]
p1 = Tree[p10,p11,p12]
p = Tree[p0,p1]
```

□

We define two trees to be equal (not isomorphic) if the corresponding lists are equal. The following declaration defines equality of trees.

```
instance Eq Utree where
 Tree(p) == Tree(q)   =  p == q
```

Isomorphism One way to check for isomorphism is to convert each tree to a normal form and compare the normal forms for equality. Normal form for a singleton tree is itself. Otherwise, a normal form is obtained by: (1) converting each subtree of the root to its normal form and (2) sorting the list of normal subtrees in ascending order. Specifically, we define the function `norm` as follows.

```
norm(Tree(p))  =  Tree(sort(map norm p))
```

Here, `map norm p` normalizes each element of list `p`. Function `sort`, described below, sorts the list and `Tree` forms a `Utree`.

Now, we can define isomorphism.

```
iso p q  =  (norm p) == (norm q)
```

Sorting In order to sort a list —any list— we must have a total order defined over its elements. For trees s and t, define $s < t$ if the list corresponding to s is lexicographically smaller than the list corresponding to t. This is a recursive definition since the elements of the lists are themselves trees.

I was surprised to see that the Haskell implementation includes lexicographic ordering over lists. So I define `<=` over trees simply by

```
instance Ord Utree where
 Tree(p) <= Tree(q)   =  p <= q
```

Any sorting algorithm can be used to sort a list of elements over which `<=` is defined. I show an insertion sort algorithm below, a relatively inefficient scheme.

```
sort[]  =  []
sort(a:x)  =  ins a (sort x)

ins a []  =  [a]
ins a (b:x)
 | a <= b  =  a:b:x
 | a >  b  =  b:(ins a x)
```

3 Putting the pieces together

```
data Utree = Tree[Utree]

instance Eq Utree where
 Tree(p) == Tree(q)  =  p == q

instance Ord Utree where
 Tree(p) <= Tree(q)  =  p <= q

norm(Tree(p))  =  Tree(sort(map norm p))

iso p q  =  (norm p) == (norm q)

sort[]  =  []
sort(a:x)  =  ins a (sort x)

ins a []  =  [a]
ins a (b:x)
 | a <= b  =  a:b:x
 | a >  b  =  b:(ins a x)
```

4 Proof of the Algorithm

We use the following notational conventions: lowercase letters —x, y, a, b— to name trees, and uppercase, X and Y, for lists.

Isomorphism We write `iso x y` as $x \sim y$. For lists X and Y, $X \approx Y$ iff the lists are of equal length and their corresponding elements are isomorphic. Next, we define these two relations formally.

(D0) $[] \approx []$

(D1) $\dfrac{a \sim b,\ X \approx Y}{a : X \approx b : Y}$

(D2) $(\mathit{Tree}\ X) \sim (\mathit{Tree}\ Y) \equiv (p\ X) \approx Y$, for some permutation p over lists.

We leave it to the reader to show that

(A1) $\sim$ and $\approx$ are equivalence relations.

(A2) $\dfrac{X \;\approx\; Y}{(Tree\ X) \;\sim\; (Tree\ Y)}$

(A3) $(Tree\ X) \;\sim\; Tree(p\ X)$, for any permutation p.

Ordering We rewrite the definition of $\leq$ over trees, where $\leq$ over lists is lexicographic ordering.

(A4) $(Tree\ X) \leq (Tree\ Y) \;\equiv\; X \leq Y$

Sort The following properties of `sort` are needed in the proof.

(A5) *sort* is a permutation over lists.

(A6) $(sort\ X) \leq (p\ X)$, for any permutation p.

The algorithm We write x^* for `norm x`, and X^* for `map norm X`. The definition of function `norm` in the new notation is

(A7) $(Tree\ X)^* = Tree(sort\ X^*)$

4.1 Theorems

The correctness of our algorithm is given by
Theorem: $x \;\sim\; y \;\;\equiv\;\; x^* = y^*$

The proof is as follows. We show that for any x, x^* is the smallest value which is isomorphic to x. That is, x and x^* are isomorphic (Lemma 1) and for isomorphic x and y, $x^* \leq y$ (Lemma 2). The theorem is proved based on these two lemmas.

Lemma 1: $x \;\sim\; x^*$
Proof: we apply induction on the structure of x. Let $x = (Tree\ X)$. Then, it is sufficient to show that

$$\langle \forall y : y \in X : y \sim y^* \rangle \;\Rightarrow\; (Tree\ X) \sim (Tree\ X)^*$$

The antecedent is $X \;\approx\; X^*$. So, we prove for all X

$$X \;\approx\; X^* \;\Rightarrow\; (Tree\ X) \;\sim\; (Tree\ X)^*$$

$$
\begin{array}{ll}
 & \mathit{Tree}\ X \\
\sim & \{\text{in (A2) replace } Y \text{ by } X^*\text{; use } X \approx X^* \text{ from the antecedent}\} \\
 & \mathit{Tree}\ X^* \\
\sim & \{\text{apply (A5) and (A3)}\} \\
 & \mathit{Tree}(\mathit{sort}\ X^*) \\
= & \{\text{apply (A7)}\} \\
 & (\mathit{Tree}\ X)^*
\end{array}
$$

Lemma 2: $x \sim y \Rightarrow x^* \leq y$
Proof: Let $x = (\mathit{Tree}\ X)$ and $y = (\mathit{Tree}\ Y)$.

$$
\begin{array}{ll}
 & x \sim y \\
= & \{\text{definitions of } x \text{ and } y\} \\
 & (\mathit{Tree}\ X) \sim (\mathit{Tree}\ Y) \\
= & \{\text{D2}\} \\
 & (p\ X) \approx Y\text{, for some permutation } p \\
\Rightarrow & \{\text{induction and the property of lexicographic ordering}\} \\
 & (p\ X)^* \leq Y\text{, for some permutation } p \\
= & \{(p\ X)^* = (p\ X^*)\text{,} \\
 & \quad \text{because } \texttt{map f (p X) = p (map f X)}\text{, for any function } \texttt{f}\} \\
 & (p\ X^*) \leq Y\text{, for some permutation } p \\
\Rightarrow & \{(\mathit{sort}\ X^*) \leq (p\ X^*)\text{, from (A6)}\} \\
 & (\mathit{sort}\ X^*) \leq Y \\
= & \{\text{A4}\} \\
 & \mathit{Tree}(\mathit{sort}\ X^*) \leq (\mathit{Tree}\ Y) \\
= & \{\text{A7 on the first term}\} \\
 & (\mathit{Tree}\ X)^* \leq (\mathit{Tree}\ Y) \\
= & \{\text{definitions of } x \text{ and } y\} \\
 & x^* \leq y
\end{array}
$$

Corollaries

C1: $x \sim y \Rightarrow x^* = y^*$

$$
\begin{array}{ll}
 & x \sim y \\
\Rightarrow & \{\text{from Lemma 1, } y \sim y^*\text{. } \sim \text{ is transitive}\} \\
 & x \sim y^* \\
\Rightarrow & \{\text{Lemma 2: replace } x \text{ by } x \text{ and } y \text{ by } y^*\} \\
 & x^* \leq y^*
\end{array}
$$

Switching the roles of x and y, $y^* \leq x^*$. Therefore, $x^* = y^*$.

C2: $x^* = y^* \Rightarrow x \sim y$

$$
\begin{array}{ll}
 & x^* = y^* \\
\Rightarrow & \{\text{from Lemma 1, } x \sim x^*\} \\
 & x \sim y^* \\
\Rightarrow & \{\text{from Lemma 1, } y^* \sim y\text{. Also, } \sim \text{ is transitive}\} \\
 & x \sim y
\end{array}
$$

Theorem: $x \ \sim \ y \quad \equiv \quad x^* = y^*$
Proof: Follows from (C1) and (C2).

Additional Corollaries

C3: $x^* \leq x$

$$\begin{array}{ll} & \textit{true} \\ \Rightarrow & \{\sim \text{ is reflexive}\} \\ & x \ \sim \ x \\ \Rightarrow & \{\text{Lemma 2}\} \\ & x^* \leq x \end{array}$$

C4: $(x^*)^* = x^*$

$$\begin{array}{ll} & \textit{true} \\ \Rightarrow & \{\text{Lemma 1}\} \\ & x^* \ \sim \ x \\ \Rightarrow & \{\text{theorem}\} \\ & (x^*)^* = x^* \end{array}$$

5 Remarks

The origin of the Problem A tree represents an expression where the internal nodes are the operators and the leaf nodes are the operands. We consider expressions in which each operator is associative and commutative; so its operands may be listed in arbitrary order. Therefore, subtrees of any node may be permuted in any manner to arrive at a tree that represents the same expression. Our algorithm decides if two trees represent the same expression, up to a reordering of the operands.

If all values in either tree are distinct, there is a simple linear algorithm for checking isomorphism. In fact, if the children of each node have distinct values then a top-down recursive algorithm can decide isomorphism. We are concerned with the case where there may be many repeated values among the children.

A Generalization: Suppose that each internal node has an associated operator and each leaf an associated operand. Then the rules of isomorphism are modified as follows.

1. A tree with a single node (the root) is isomorphic only to a tree with a single node that has the same associated operand.

2. Two trees with roots A and B, none of which is a single-node tree, are isomorphic iff the associated operators at the roots are identical and there is a 1-1 correspondence between the subtrees of A and of B such that the corresponding subtrees are isomorphic.

Develop an isomorphism algorithm for this case.

A further generalization is as follows. Designate a subset of operators as *commutative and associative*. Only the subtrees of such operators may be reordered. Additionally, some of the operators can be designated *idempotent*; any subtree of such an operator may be removed if it is isomorphic to another subtree (of this operator). An operator may be commutative and associative as well as idempotent. Develop isomorphism algorithm for this case.

Haskell I had earlier written this program in CAML, a derivative of ML. The Haskell version is shorter and more elegant. First, a sophisticated type system helped in designing the appropriate data structure. And, the type classes of Haskell made it particularly easy to define equality and a total order over trees.

Acknowledgment I am grateful to Ham Richards who answered all my questions about Haskell promptly, cheerfully, and correctly. Perceptive comments by C.A.R. Hoare have helped improve the presentation. Edsger W. Dijkstra's insistence on rigor has led to a simpler (and more rigorous) proof.