Lecture VII

Keywords:
recursive datatypes: lists, trees, \(\lambda\) calculus; tree manipulation; tree listings: preorder, inorder, postorder; tree exploration: breadth-first and depth-first search; polymorphic exceptions; isomorphisms.

References:
[MLWP, Chapter 4]

Recursive datatypes

Datatype definitions, including polymorphic ones, can be recursive.

The built-in type operator of \textit{lists} might be defined as follows:

\[
\text{infixr 5 :: ;}
\]
\[
\text{datatype 'a list}
\]
\[
= \text{nil} \mid :: \text{of } 'a \ast 'a \text{ list} ;
\]

In the same vein, the polymorphic datatype of (planar) \textit{binary trees} with nodes where data items are stored is given by:

\[
\text{datatype 'a tree}
\]
\[
= \text{empty} \mid \text{node of } 'a \ast 'a \text{ tree} \ast 'a \text{ tree} ;
\]

Semantics

The set \(\text{Val(}\tau \text{ list})\) of \textit{values} of the type \(\tau \text{ list}\) is inductively given by the following rules:

\[
\text{nil} \in \text{Val}(\tau \text{ list})
\]
\[
\nu \in \text{Val}(\tau) \quad \ell \in \text{Val}(\tau \text{ list})
\]
\[
\nu :: \ell \in \text{Val}(\tau \text{ list})
\]

That is, \(\text{Val}(\tau \text{ list})\) is the smallest set containing \texttt{nil} and closed under performing the operation \(\nu :: \_\) for values \(\nu\) of type \(\tau\).

Question: What is the set of values of \(\tau \text{ tree}\)?

Further recursive datatypes

Examples:
1. Non-empty planar finitely-branching trees and forests.
   (a) Recursive version.
   \[
   \text{datatype}
   \]
   \[
   'a \text{ FBtree} = \text{node of } 'a \ast 'a \text{ FBtree list} ;
   \]
   \[
   \text{type}
   \]
   \[
   'a \text{ FBforest} = 'a \text{ FBtree list} ;
   \]
   (b) Mutual-recursive version.
   \[
   \text{datatype}
   \]
   \[
   'a \text{ FBtree} = \text{node of } 'a \ast 'a \text{ FBforest}
   \]
   \[
   \text{and}
   \]
   \[
   'a \text{ FBforest} = \text{forest of } 'a \text{ FBtree list} ;
   \]
   Question: What are the set of values of \(\tau \text{ FBtree}\) and \(\tau \text{ FBforest}\)?
2. \(\lambda\) calculus.

```lisp
datatype
  D = f of D -> D ;

NB: It is non-trivial to give semantics to \(D\). This was done
by Dana Scott in the early 70’s, and gave rise to *Domain Theory*.

References:
♦ D. Scott. Continuous lattices. In *Toposes, Algebraic
  Geometry and Logic*, pages 97–136, Lecture Notes
  in Mathematics 274, 1972.
♦ D. Scott. A type-theoretical alternative to ISWIM, CUCH,
  OHWI. Unpublished notes, Oxford University, 1969.
  (Published in *Theoretical Computer Science*,
  121(1-2):411–440, 1993.)
```

4. \(\text{datatype}
```
  | dir = L | R ;
```
  | exception E ;
```

```lisp
| fun subtree [] t = t
|   | subtree ( L::D ) ( node(_,l,_) )
|   |     = subtree D l
|   | subtree ( R::D ) ( node(_,_,r) )
|   |     = subtree D r
| subtree _ _
|     = raise E ;
```
Inorder without append

fun inorder t
  = let
    fun accinorder acc empty = acc
    | accinorder acc ( node(n,l,r) )
      = accinorder ( n :: accinorder acc r ) l
  in
    accinorder [] t
  end;

  inorder( node(3,node(2,node(1,empty,empty), empty),
              node(4,empty,
                   node(5,empty,empty))) );

val it = [1,2,3,4,5] : int list

Tree exploration

1. DFS without append

fun dfs' P t
  = let
    fun auxdfs( node(n,F) )
      = if P n then SOME n
         else auxdfs( F @ T )
  in
    auxdfs [t]
  end;

val dfs' = fn : ('a -> bool) -> 'a FBtree -> 'a option

Tree exploration

Breadth-first search

datatype
  'a FBtree = node of 'a * 'a FBtree list ;

fun bfs P t
  = let fun auxbfs [] = NONE
         | auxbfs( node(n,F)::T )
            = if P n then SOME n
               else auxbfs( T @ F )
  in
    auxbfs [t] end ;

val bfs = fn : ('a->bool) -> 'a FBtree -> 'a option


2. DFS without append

fun dfs P t
  = let
    fun auxdfs( node(n,F) )
      = if P n then SOME n
         else fold1
             ( fn(t,r) => case r of
               NONE => auxdfs t | _ => r )
             ( fn(t,r) => case r of
               NONE => auxdfs t | _ => r )
  end;

val dfs = fn : ('a -> bool) -> 'a FBtree -> 'a option
3. DFS without append; raising an exception when successful.

```ml
fun dfs0 P (t: 'a FBtree) =
    let
        exception Ok of 'a;
        fun auxdfs( node(n,F) ) =
            if P n then raise Ok n
            else foldl (fn(t,_) => auxdfs t) NONE F ;
    in
        auxdfs t handle Ok n => SOME n end;
    val dfs0 = fn :
        ('a -> bool) -> 'a FBtree -> 'a option
```

**Warning:** When a *polymorphic exception* is declared, ML ensures that it is used with only one type. The type of a top level exception must be monomorphic and the type variables of a local exception are frozen.

Consider the following nonsense:

```ml
exception Poly of 'a ; (**ILLEGAL!!!**) (raise Poly true) handle Poly x => x+1 ;
```

---

**Further topics**

Quite surprisingly, there are very sophisticated *non-recursive* programs between recursive datatypes.

**References:**