Funktionale Konstruktion des Suffixbaums

Man kann den Suffix-Trie oder -Baum eines Texts auch funktional berechnen. (R.Giegerich, S.Kurtz, ESOP'94, LNCS 788.)

Statt *on-line*, d.h. inkrementell beim Lesen des Texts, kann man den Baum auch *lazy*, d.h. inkrementell bei der Benutzung, konstruieren. Die Grundidee ist:

(i) konstruiere den Baum von der Wurzel zu den Blättern, beginnend mit der Suffixmenge des Text an der Wurzel;

(ii) zerlege die Suffixmenge S an einem Knoten nach

$$aw_a \cdot S_a \;\cup\; bw_b \cdot S_b \;\cup\; \ldots$$

nach längsten gemeinsamen Präfixen aus $\Sigma\Sigma^*$;

(iii) markiere die Kante zum a-Nachfolger mit aw_a und konstruiere rekursiv aus S_a den Teilbaum bei diesem Knoten;

(iv) baue den Baum bei einem Knoten erst dann, wenn eine Suche den Knoten passiert.

Bem. Ergibt eine sehr einfache Konstruktion, die im schlimmsten Fall (bei $a^{n-1}\$$) $O(|\Sigma| \cdot |t|^2)$ Schritte braucht, aber im Durchschnitt $O(|\Sigma| \cdot |t| \cdot \log |t|)$.

Eine imperative Implementierung ist laut Giegerich/Kurtz (bis $|t| = 100.000$) 5 mal schneller als eine Implementierung des Verfahrens von Ukkonen.

Verzögern/Erzwingen des Baumaufbaus

In der *call-by-value*-Auswertung von SML simulieren wir die verzögerte Auswertung mit `delay/force` (nicht nötig in Haskell).

⟨*lazySuffixTree.sml*⟩≡

```
functor LazySufTree(Set:SET) =
 struct
   datatype 'a sTree =
            Leaf
          | Branch of ('a list * ('a sTree) ref) list
          | Delayed of (unit -> 'a sTree)

   type 'a edgeFunction =
        'a list list -> 'a list * (('a list) list)

   fun delay st = ref (Delayed st)
   and force str = case !str of
           Delayed f => let val st = f()
                        in str := st; st end
         | st => st

   fun find Leaf (p::ps) = NONE
     | find (Branch ((b::bs,st)::br)) (p::ps) =
       if p = b then find (Branch [(bs,st)]) ps
       else find (Branch br) (p::ps)
     | find (Branch (([],st)::br)) ps =
       find (force st) ps  (* Baum entfalten *)
     | find (Branch []) ps = NONE
     | find (Delayed f) ps = find (f()) ps   (* top-level *)
     | find tree [] = SOME tree
```

Längstes gemeinsames Präfix

⟨*lazySuffixTree.sml*⟩+≡

```
(* Longest common prefix of two lists *)

fun lcp [] ys = ([],[],ys)
  | lcp xs [] = ([],xs,[])
  | lcp (x::xs) (y::ys) =
    if x = y then
        let val (aa,bs,cs) = (lcp xs ys)
        in (x::aa,bs,cs) end
    else ([],x::xs,y::ys)

(* Longest common prefix of a list of lists,
   and the rem.suffixes *)

fun Lcp [] (Prefix,Suffs) = (Prefix,Suffs)
  | Lcp (v::L) ([],_) = Lcp L ([v],[[]])
  | Lcp (v::L) ([w],Suffs) =
    let
        val (p,vs,ws) = lcp v w
        fun precat x s = x @ s
    in
        Lcp L ([p],vs :: (map (precat ws) Suffs))
    end
```

Das ist nicht auf Effizienz angelegt.

Rekursive Top-Down-Konstruktion des Baums

aus einer Funktion `suf` zur Berechnung der Suffixe eines Texts, seinem `alphabet`, einer Kantenbildung `edge` und dem `text`:

⟨*lazySuffixTree.sml*⟩+≡

```
fun sufTree (suf,alphabet) edge text =
  let
    fun chopLeft a ss =
        map tl (List.filter
                   (fn (s::_) => s=a | _ => false) ss)

    fun sTr [[]] = Leaf
      | sTr ss = let
            fun branch a =
                let
                    val E = edge (chopLeft a ss)
                in
                    case E of
                        (_,y::ys) => SOME (a,E)
                      | _ => NONE
                end
            fun mkSubtree (a,(p,ys)) =
                (a::p, delay (fn () => sTr ys))  (* ! *)
        in
            Branch (map mkSubtree
                        (List.mapPartial branch alphabet))
        end
  in
      sTr (suf text)
  end
```

⟨*lazySuffixTree.sml*⟩+≡

```
fun suffixTree edge text =
    let
        val alphabet = Set.list (Set.mk text)
        fun suffixes [] = [] (* non-empty suffixes *)
          | suffixes (x::xs) = (x::xs) :: suffixes xs
    in
        sufTree (suffixes,alphabet) edge text
    end
```

In der Struktur `Set` werden endliche Mengen durch Listen ohne mehrfach vorkommende Elemente repräsentiert. (Übung!)

Kantenbildung für Suffix-Trie bzw. -Tree

Um einen Suffix-Trie zu erhalten, wird (außer dem schon isolierten ersten Buchstaben) nichts aus den Restwörtern extrahiert, für einen Suffix-Baum das längste gemeinsame Präfix:

⟨*lazySuffixTree.sml*⟩+≡

```
fun mkTrie text = suffixTree (fn ss => ([],ss))

fun mkSuffixTree text =
    let
        fun edge ss =
            case (Lcp ss ([],[])) of
                ([p],Suffs) => (p,Suffs)
              | (Pref,Suffs) => ([],Suffs) (* nie *)
    in
        suffixTree edge text
    end
```

Graphische Darstellung (`.dot`-Format)

⟨*lazySuffixTree.sml*⟩+≡

```
fun draw st filename =
   let
      val stream = TextIO.openOut (filename ^ ".dot")
      fun write s = TextIO.output (stream, s)
      fun writeLabel w = implode w (* w: char list *)
      fun node n = Int.toString n

      fun drawTree (Branch tr) n = drawBranches tr n n
        | drawTree Leaf n = n
        | drawTree (Delayed f) n = n
      and drawBranches ((w,tw)::branches) n k =
          let
              val m = drawTree (!tw) (k+1)
          in
              (write (node n ^ " -> " ^ (node (k+1)));
               write (" [label = " ^ writeLabel w ^ "] ;\n");
               (case (!tw) of (Delayed f) =>
                  write (node (k+1) ^ " [shape=triangle]\n")
                            | _ => ());
               drawBranches branches n m)
          end
        | drawBranches [] n k = k (* k: verbrauchte Nr. *)
   in
       write ("digraph " ^ filename
         ^ " {center=true;\nsize=\"7.5,14\";\
         \ \nratio=compress;\nnode [shape=plaintext];\n");
       drawTree st 0; write "}\n";
       TextIO.closeOut stream
   end
```

Graphische Anzeige

⟨*lazySuffixTree.sml*⟩+≡

```
    fun showgraph filename =
        (OS.Process.system
             ("dot -Tps " ^ filename
              ^ ".dot -o "^ filename ^".ps ");
         OS.Process.system
             ("gv " ^ filename ^ ".ps &"))

    fun show st =
        (draw st "suffixTreeTmp";
         showgraph "suffixTreeTmp")
   end
```

Modul LazySuffixTree

⟨*lazySuffixTree.sml*⟩+≡

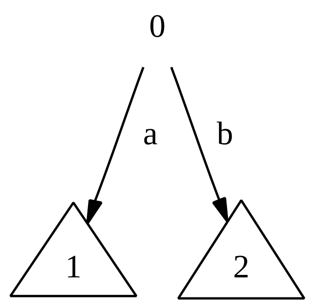
```
  signature LAZY_SUFFIX_TREE =
      sig
          datatype 'a sTree =
              Leaf
            | Branch of ('a list * ('a sTree) ref) list
            | Delayed of (unit -> 'a sTree)

          val mkTrie : ''a list -> ''a sTree
          val mkSuffixTree : ''a list -> ''a sTree
          val find : ''a sTree-> ''a list -> ''a sTree option
          val show : char sTree -> OS.Process.status
      end

  structure LazySuffixTree =
     LazySufTree(Set) : LAZY_SUFFIX_TREE
```

Beispiel: Verzögerter Aufbau des Suffixbaums für `abaabb`

Anzeige: nicht besuchte Teilbäume sind als Dreiecke dargestellt.

⟨*Baum für abaabb nach dem Aufbau (Knoten 0 besucht)*⟩≡

```
- val st = mkSuffixTree (explode "abaabb");
val st = Branch [([#"a"],ref (Delayed #)),
                 ([#"b"],ref (Delayed #))] : char sTree
- show st;
```



⟨*Baum für abaabb nach der Suche nach a (Knoten 1 besucht)*⟩≡

```
- find st #"a";
val it = SOME (Branch [([#,#,#],ref #),([#],ref #)])
- show st;
```