SML-Kurs, SS 2003,
Lösungen zu ausgewählten Aufgaben

Hans Leiß

5. Juni 2003

Lösung zu Aufgabe ??: Die Signatur `TRIE` soll einen Typ von Bäumen enthalten, an dessen Kanten Objekte vom Typ `'a` stehen und an dessen Knoten Objekte vom Typ `'b` (Informationen) stehen (oder zumindest stehen *können*. Im allgemeinen wird nicht bei jedem inneren Knoten eine Information stehen.)

⟨*trie.sml*⟩≡

```
signature TRIE =
    sig
        type ('a,'b) t
        val empty : ('a,'b) t
        val insert : (''a,'b) t -> (''a list * 'b) -> (''a,'b) t
        val lookup : (''a,'b) t -> ''a list -> 'b option
    end
```

Statt `('a,'b) t` hätten wir auch `('a,'b) trie` sagen können. △

Lösung zu Aufgabe ??:

(i) Die Struktur sollte an den Knoten eine `'b option` haben, damit man auch Knoten haben kann, wo keine Information steht. Blattknoten sind die Verzweigungsknoten, bei denen eine leere Liste von direkten Teilbäumen auftritt.

⟨*trie.sml*⟩+≡
```
structure Trie =
    struct
        datatype ('a,'b) t =
            Br of ('b option) * ('a * ('a,'b) t) list

        val empty = Br(NONE,[])

        fun insert (Br(info,ts)) (c::cs,neu_info) =
            Br(info,insert_tries ts (c::cs,neu_info))
          | insert (Br(info,ts)) ([],neu_info) =
            Br(SOME neu_info,ts)
        and insert_tries ((x,tr)::ts) (c::cs,neu_info) =
            if c = x then (x,(insert tr (cs,neu_info))) :: ts
            else (x,tr)::(insert_tries ts (c::cs,neu_info))
          | insert_tries [] (c::cs,neu_info) =
                [(c,insert empty (cs,neu_info))]

        fun lookup (Br(info,ts)) [] = info
          | lookup (Br(info,ts)) (c::cs) =
            lookup_tries ts (c::cs)
        and lookup_tries ((x,tr)::ts) (c::cs) =
            if x = c then lookup tr cs
            else lookup_tries ts (c::cs)
          | lookup_tries ((x,tr)::ts) [] = NONE (* Sollte Ausnahme sein! *)
          | lookup_tries [] (c::cs) = NONE
    end
```

(ii) Ein Beispiel erhalten wir durch

⟨*trie.sml*⟩+≡
```
val tr0 = Trie.insert Trie.empty ([1,2,3],"Aachen");
val tr  = Trie.insert tr0 ([1,2,4],"Augsburg");

Compiler.Control.Print.printDepth := 30;
```

Das ergibt den Trie

⟨*Wert von tr*⟩≡

```
- val tr  = Trie.insert tr0 ([1,2,4],"Augsburg");
val tr =
  Br
    (NONE,
     [(1,
       Br
         (NONE,
          [(2,
            Br
              (NONE,
               [(3,Br (SOME "Aachen",[])),(4,Br (SOME "Augsburg",[]))]))]))])
  : (int,string) Trie.t
-
```

Fügt man jetzt noch `([1,2],"Bonn")` hinzu, erhält man:

⟨*Erweiterung von tr*⟩≡

```
- val tr = Trie.insert tr ([1,2],"Bonn");

val tr =
  Br
    (NONE,
     [(1,
       Br
         (NONE,
          [(2,
            Br
              (SOME "Bonn",
               [(3,Br (SOME "Aachen",[])),(4,Br (SOME "Augsburg",[]))]))]))])
  : (int,string) Trie.t
```

Die Suche ergibt z.B.

⟨*Ausgabe von gesuchter Information*⟩≡

```
- Trie.lookup tr [1];
val it = NONE : string option
- Trie.lookup tr [1,2];
val it = SOME "Bonn" : string option
-
```

△

Lösung zu Aufgabe ??:

⟨*Definition einer neuen Struktur durch opake Einschränkung auf die Signatur*⟩≡

```
- structure absTrie = Trie :> TRIE;
structure absTrie : TRIE
```

⟨*Vergleich*⟩≡

```
- Trie.empty = absTrie.empty;
stdIn:1.1-38.11 Error: operator and operand don't agree [tycon mismatch]
  operator domain: (''Z,''Y) Trie.t * (''Z,''Y) Trie.t
  operand:         (''Z,''Y) Trie.t * ('X,'W) absTrie.t
  in expression:
    Trie.empty = absTrie.empty

- absTrie.insert absTrie.empty ([1,2],"Bonn");
val it = - : (int,string) absTrie.t

- absTrie.lookup it [1,2];
val it = SOME "Bonn" : string option
```

Durch die opake Einschränkung wird die Implementierung von `absTrie.t` dem Anwender verborgen. Er kann nur `insert`, `lookup` und `empty` verwenden, um Suchbäume vom Typ `absTrie.t` aufzubauen und zu verwenden. △