Referenzierbare Objekte

SML hat Speicherzellen (Zeiger), aber keine Zeigerarithmetik.

Der *Typkonstruktor* `ref` konstruiert aus einem Typ τ den Typ τ `ref` der *Zellen zur Speicherung von τ-Objekten*:

$\langle$*expl*$\rangle\equiv$

```
type meineZellen = (int * bool) ref
```

Der *Wertkonstruktor* `ref : 'a -> 'a ref` bildet eine Zelle, in der der Wert seines Argument gespeichert ist:

$\langle$*expl*$\rangle{+}\equiv$

```
- ref (3,false);
val it = ref (3,false) : (int * bool) ref
```

Meist gibt man den Zellen einen Namen:

$\langle$*expl*$\rangle{+}\equiv$

```
val zelle = ref (3,false) : meineZellen
```

Jede Auswertung eines Terms `(ref t)` erzeugt eine neue Zelle:

$\langle$*expl*$\rangle{+}\equiv$

```
- ref (3,false) = ref (3,false);
val it = false : bool
```

Zellen sind durch = vergleichbar, auch wenn es die Inhalte nicht sind:

$\langle$*expl*$\rangle{+}\equiv$

```
- ref (fn x => x+1) = ref (fn x => x+1);
val it = false : bool
```

Referenzierbare Objekte: Zuweisung und Ablesen

Den Inhalt einer Zelle kann man mit dem *Zuweisungsoperator* `:= : 'a ref * 'a -> unit` ändern, falls die zu speichernden Werte den richtigen Typ haben:

$\langle$*expl*$\rangle{+}{\equiv}$

```
- zelle := (4,true);
val it = () : unit
- zelle := (true,4);
Error: operator and operand don't agree [tycon mismatch]
```

Durch die *Dereferenzierung* `! : 'a ref -> 'a` kann man den Inhalt einer Zelle lesen:

$\langle$*expl*$\rangle{+}{\equiv}$

```
- ! zelle
val it = (4,true) : int * bool
```

Wertbeschränkung

Freie Variablen im Typ eines zu speichernden Objekts werden bei der Bildung einer Zelle eingeschränkt:

⟨*expl*⟩+≡

```
- val dummyFunktion = ref (fn x => x);
Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val dummyFunktion = ref fn : (?.X1 -> ?.X1) ref
```

Diese Zelle kann man nur mit Werten beschreiben, die den (unbekannten) Typ nicht ändern:

⟨*expl*⟩+≡

```
- dummyFunktion := (fn x => (2 * 4; x));
val it = () : unit
- dummyFunktion := (fn x => 2);
Error: operator and operand don't agree [literal]
  operator domain: (?.X1 -> ?.X1) ref * (?.X1 -> ?.X1)
  operand:         (?.X1 -> ?.X1) ref * (?.X1 -> int)
  in expression:   dummyFunktion := (fn x => 2)
```

Umweg: In Funktion einbetten

⟨*expl*⟩+≡

```
- val leerZelle = ref [];
Warning: type vars not generalized ...
val leerZelle = ref [] : ?.X1 list ref

- fun erzeugeLeerzelle () = ref [];
val erzeugeLeerzelle = fn : unit -> 'a list ref
val intlistZelle : int list ref = erzeugeLeerzelle ();
```

Beispiel: Zyklisch verketteter Puffer (L.Paulson, Chap. 8)

Wir wollen einen Puffer mif folgenden Eigenschaften bilden:

1. Der Puffer besteht aus einem Ring von Speicherzellen.
2. Jede Zelle enthält ein Objekt vom Typ `'a`.
3. Der Inhalt einer Zelle kann gelesen werden.
4. Jede Zelle kann entfernt werden, wobei ihre linken und rechten Nachbarn zu direkten Nachbarn werden.
5. Zwischen zwei benachbarte Zellen kann eine neue Zelle mit einem Objekt vom Typ `'a` eingefügt werden.
6. Von jeder Zelle können wir die nächste links und die nächste rechts liegende Zelle erreichen.

⟨*buffer*⟩≡

```
signature RINGBUF =
  sig
    eqtype 'a t
    exception Empty
    val empty: unit -> 'a t
    val null: 'a t -> bool
    val label: 'a t -> 'a
    val moveLeft: 'a t -> unit
    val moveRight: 'a t -> unit
    val insert: 'a t * 'a -> unit
    val delete: 'a t -> 'a
  end;
```

Zyklisch verketteter Puffer (Implementierung)

⟨*buffer*⟩+≡

```
structure RingBuf :> RINGBUF =
  struct
    datatype 'a buf =
      Nil
    | Node of 'a buf ref * 'a * 'a buf ref;

    datatype 'a t = Ptr of 'a buf ref;

    exception Empty;

    fun left (Node(lp,_,_)) = lp
      | left Nil = raise Empty;
    fun right (Node(_,_,rp)) = rp
      | right Nil = raise Empty;

    fun null (Ptr p) = case !p of
      Nil => true | Node(_,x,_) => false;

    fun empty() = Ptr(ref Nil); (* neuer(!) Puffer *)

    fun label (Ptr p) = case !p of
      Nil => raise Empty | Node(_,x,_) => x;

    fun moveLeft (Ptr p) = (p := !(left(!p)));
    fun moveRight (Ptr p) = (p := !(right(!p)));
```

Puffer: Einfügen und Löschen

⟨*buffer*⟩+≡

```
  fun insert (Ptr p, x) =
    case !p of
      Nil => (* Zelle mit sich selbst als Nachbarn *)
        let val lp = ref Nil
            and rp = ref Nil
            val new = Node(lp,x,rp)
        in
          lp := new;  rp := new;  p := new
        end
    | Node(lp,_,_) => (* neue Zelle links einfuegen *)
          let
            val new = Node(ref(!lp), x, ref(!p))
          in
            right(!lp) := new;  lp := new
          end;

  fun delete (Ptr p) =
    case !p of
        Nil => raise Empty
      | Node(lp,x,rp) =>
           (if left(!lp) = lp (* einzige Zelle *)
              then p := Nil
            else (right(!lp) := !rp;
                  left (!rp) := !lp;
                  p := !rp);
            x) (* Zelleninhalt als Ergebnis *)
end;
```

Puffer: Anwenden

⟨*buffer*⟩+≡

```
open RingBuf;

- val B = empty();
Warning: type vars not generalized ...
val B = - : ?.X1 t
```

Das ist nutzlos, da man keine Objekte des *unbekannten* Typs `?.X1 t` bilden kann! Man muß den Typ der zu speichernden Objekte bei der Erzeugung des Puffers angeben:

⟨*expl*⟩+≡

```
- val B : int t = empty();
val B : int t
```

Jetzt kann man Objekte in den Puffer einfügen:

⟨*expl*⟩+≡

```
- insert (B,0); insert (B,1); insert (B,2);
```

Da die Objekte in neue, links angefügte Zellen geschrieben wurden, hat sich der Inhalt der Hauptzelle `B` nicht geändert:

⟨*expl*⟩+≡

```
label B;
val it = 0 : int
```

Die linke Nachbarzelle enthält den zuletzt eingefügten Wert:

⟨*expl*⟩+≡

```
moveLeft B; label B;
- val it = 2 : int
```

Puffer: Zellen löschen

Löschen der Zelle (mit Übergang zum rechten Nachbarn) gibt:

$\langle expl \rangle{+}\equiv$

```
- delete B; label B;
val it = 0 : int
```

Die linke Nachbarzelle enthält jetzt die `1`,

$\langle expl \rangle{+}\equiv$

```
- moveLeft B; label B;
val it = 1 :int
```

und deren linker Nachbar ist die ursprüngliche Hauptzelle:

$\langle expl \rangle{+}\equiv$

```
- moveLeft B; label B;
val it = 0 : int
```

Durch Übergänge zu linken Nachbarn kann man also wieder zur Ausgangszelle. (Im allgemeinen kann man die Zelle natürlich an ihrem Inhalt nicht eindeutig erkennen!)