Funktoren mit abhängigen Ergebnissignaturen

In manchen Fällen liefert *weder eine transparente noch eine opake* Signatureinschränkung die gewünschte Spezifikation eines Funktors.

⟨*beisp*⟩≡

```
functor Matrix(R : HALBRING) : MATRIX =
   struct
      structure Ring = R
      type r = Ring.t
      type t = r vector vector
      ...
   end
```

Durch die transparente Einschränkung `:MATRIX` wird leider der Implementierungstyp `t` der Matrizen sichtbar. Er wird bei der opaken Einschränkung in

⟨*beisp*⟩+≡

```
functor Matrix(R : HALBRING) :> MATRIX = ...
```

abstrakt, aber gleichzeitig wird *auch der Implementierungstyp* `r` *der Ringelemente* abstrakt, was man nicht brauchen kann.

Man braucht Funktoren, *deren Ergebnissignatur(!) von Typen in ihrer Argumentstruktur(!) abhängt*:

⟨*beisp*⟩+≡

```
functor Matrix'(R:HALBRING) =
   struct
     ...
   end :> MATRIX where type Ring.t = R.t
```

```
SIGNATURE where type t = s
```

Allgemein kann man eine Signatur im Nachhinein spezialisieren, wie in

⟨*beisp*⟩+≡
```
signature IntMATRIX = MATRIX where type Ring.t = int
```

Damit kann man z.B. eine transparentes Matrixpaket einschränken:

⟨*beisp*⟩+≡
```
structure IntMatTransp =
   Matrix(IntRing : HALBRING) : MATRIX
structure IntMat = IntMatTransp :> IntMATRIX
```

In `IntMat` ist der Typ `r` transparent, aber der Typ `t` der Matrizen nicht.

Bei der Einschränkung eines Signaturausdrucks `SIG` durch

⟨*sig where*⟩≡
```
SIG where type A.t = B.s
```

muß der Typname `A.t` im Wert von `SIG` eingeführt werden, während der Typ `B.s` unabhängig vom Wert von `SIG` sein muß.

Kennt man `B.s` nur aus einem Funktorargument, wie in

⟨*beisp*⟩+≡
```
functor Matrix'(R:HALBRING) =
   struct
     ...
   end :> MATRIX where type Ring.t = R.t ,
```

so heißt `SIG where type A.t = B.s` eine *abhängige Signatur*.