Funktionale Programmierung

Generische Programmierung

Hans-Wolfgang Loidl, Andreas Abel

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

25. Juni 2009

Grundlagen

Sprachdesign ist grundsätzlich ein Kompromiss zwischen **Flexibilität** und **Sicherheit**.

Sprachen mit statischem Typ-System bieten einen hohen Grad an Sicherheit ("well-typed programs don't go wrong"), sind aber weniger flexibel als ungetypte Sprachen.

Ein Ansatz um die Flexibilität zu erhöhen ist es typ-basierte Techniken einzuführen, um gleichartige, **generische**, Programme zu definieren.

Arten von Generischen Programmen

Der Begriff "generisch" in Programmiersprachen ist sehr generisch!

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Wert: Argument einer Funktion

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Typ: Polymorphismus, in 2 Formen:

- Parametrischer Polymorphismus: der gleich Code wird für verschiedene Typen verwendet z.B: length Funktion

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \quad\ = 0 \\
&\mathit{length}\ (\_ : \mathit{xs}) = 1 + (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

- Ad-hoc Polymorphismus (overloading): der Code hängt vom Typ selbst ab; z.B. Addition auf Int, Complex, etc

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Funktion: damit erhalten wir Funktionen höherer Ordnung; z.B:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] \quad\ = [\,] \\
&map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)
\end{aligned}
$$

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Interface: man abstrahiert über eine Menge von typ-basierten Eigenschaften, die für alle Funktionen gelten müssen; z.B: Typ-Context in Haskell Typ-Klassen
instance $(Eq\ a) \Rightarrow Eq\ [a]$ **where**...

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Eigenschaft: als Erweiterung von Interfaces, abstrahiert man über generelle Eigenschaften die für die Funktionen gelten müssen; z.B: Monaden-Regeln für Instanzen der Monad Klasse (muss manuell bewiesen werden):

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= k = k\ x \\
&k \gg\!= \mathit{return} \quad = k \\
&(m \gg\!= \lambda x \to k\ x) \gg\!= k' = m \gg\!= \lambda x \to (k\ x \gg\!= k')
\end{aligned}
$$

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:
Phase: man unterteilt die Compilierung in Phasen, wobei frühe Phasen selbst wieder Programme erzeugen ("meta-programming");
z.B: Verwendung eines Parser Generators.
In Template Haskell kann man Projektionen auf ein Tupel anstatt

$$fst3 = \lambda(x, \_, \_) \rightarrow x$$

wie folgt definieren:

```
sel :: Int → Int ExpQ
sel i n = lamE [pat] body
          where pat  = tupP (map varP vars)
                body = varE (vars !! (i − 1))
                vars = [mkName ("a" ++ show j) | j ← [1 .. n]]
fst3 = $(sel 1 3)
```

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Struktur ("shape"): abstrahiert über die Daten-Struktur; damit wird es möglich Code zu schreiben der z.B. sowohl auf Listen als auch auf Bäumen funktioniert (*map*)

Scrap Your Boilerplate (SYB)

Leichtgewichtige generische Programmierung in Haskell.

Es eine Menge von primitiven Operationen definiert, die mittels **deriving** automatisch erzeugt werden können.

Verwendet folgende Haskell Erweiterungen:

- **rank-2 Polymorphismus**
- **type-safe type cast**

Beispiel: Arithmetische Ausdrücke

$$
\begin{aligned}
\textbf{data}\ \mathit{ArithExpr}\ a = {} & \mathit{Var}\ \mathit{String} \\
& \mid \mathit{Lit}\ a \\
& \mid \mathit{ArithExpr}\ a : + \mathit{ArithExpr}\ a \\
& \mid \mathit{ArithExpr}\ a : - \mathit{ArithExpr}\ a \\
& \mid \mathit{ArithExpr}\ a : * \mathit{ArithExpr}\ a \\
& \mid \mathit{ArithExpr}\ a : / \mathit{ArithExpr}\ a
\end{aligned}
$$

In standard Haskell müssen wir viel "boilerplate" Code schreiben, der die Datenstruktur traversiert:

$$
\begin{aligned}
&\mathit{inc} :: \mathit{ArithExpr}\ \mathit{Int} \to \mathit{ArithExpr}\ \mathit{Int} \\
&\mathit{inc}\ (\mathit{Var}\ s) = \mathit{Var}\ s \\
&\mathit{inc}\ (\mathit{Lit}\ x) = \mathit{Lit}\ (x + 1) \\
&\mathit{inc}\ (e1 : + e2) = (\mathit{inc}\ e1) : + (\mathit{inc}\ e2) \\
&\mathit{inc}\ (e1 : - e2) = (\mathit{inc}\ e1) : - (\mathit{inc}\ e2) \\
&\mathit{inc}\ (e1 : * e2) = (\mathit{inc}\ e1) : * (\mathit{inc}\ e2) \\
&\mathit{inc}\ (e1 : / e2) = (\mathit{inc}\ e1) : / (\mathit{inc}\ e2)
\end{aligned}
$$

Beispiel: Arithmetische Ausdrücke

Mittels Scrap Your Boilerplate (SYB), können wir kurz schreiben:

$$inc :: Int \to Int$$
$$inc\ x = x + 1$$

$$increment :: Data\ a \Rightarrow a \to a$$
$$increment = everywhere\ (mkT\ inc)$$

Beispiel: Arithmetische Ausdrücke

Mittels Scrap Your Boilerplate (SYB), können wir kurz schreiben:

$$inc :: Int \to Int$$
$$inc\ x = x + 1$$

$$increment :: Data\ a \Rightarrow a \to a$$
$$increment = everywhere\ (mkT\ inc)$$

D.h. man unterscheidet 3 Code-Teile:

- **Arbeiter-Code:** kurzer Code der die Essenz der Transformation ausdrückt;
- **Boilerplate-Code:** automatisch generierter Code für Traversierung der Datenstruktur;
- **Bibliothek:** Kombinatoren zum Verknüpfen beider Teile

Im Gegensatz zu anderen Ansätzen zu generischer Programmierung, wird die interne Struktur des Typs nie explizit gemacht.

Design von SYB

Zentral für das Design von SYB ist folgende Typ-Klasse:

class *Typable a* **where**
  typeOf :: $a \rightarrow$ *TypeRep*

GHC kann für allgemeine algebraische Datentypen Instanzen dieser Klasse automatisch erzeugen.

Die Typrepräsentation für jeden Typ ist garantiert eindeutig.

Damit kann man im Code explizit einen Typvergleich durchführen.

Basics: Data Klasse

Die *Typable* Klasse ist die unterste Ebene zur Unterstützung von generischer Programmierung.

Darauf aufbauend werden generische Abstraktionen definiert, z.B. ein generisches *gmapT* und ein generisches *gfoldl*:

$$
\begin{aligned}
&\textbf{class } \mathit{Typeable}\ d \Rightarrow \mathit{Data}\ d \textbf{ where} \\
&\quad \mathit{gmapT} :: (\mathit{forall}\ b\ \mathit{Data}\ b \Rightarrow b \to b) \to a \to a \\
&\quad \mathit{gfoldl} \quad :: (\mathit{forall}\ a\ b\ \mathit{Data}\ a \Rightarrow c\ (a \to b) \to a \to c\ b) \\
&\qquad\qquad \to (\mathit{forall}\ g\ g \to c\ g) \\
&\qquad\qquad \to d \\
&\qquad\qquad \to c\ d
\end{aligned}
$$

Basics: Data Klasse

Eine Instanz von *Data* für Listen kann wie folgt definiert werden:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Typeable}\ a, \mathit{Data}\ a) \Rightarrow \mathit{Data}\ (\mathit{List}\ a)\ \textbf{where}\\
&\quad \mathit{gmapT}\ f\ \mathit{Nil} &&= \mathit{Nil}\\
&\quad \mathit{gmapT}\ f\ (\mathit{Cons}\ x\ xs) &&= \mathit{Cons}\ (f\ x)\ (f\ xs)\\
&\quad \mathit{gfoldl}\ k\ z\ \mathit{Nil} &&= z\ \mathit{Nil}\\
&\quad \mathit{gfoldl}\ k\ z\ (\mathit{Cons}\ h\ t) &&= (z\ \mathit{Cons}\ `k`\ h)\ `k`\ t
\end{aligned}
$$

Beispiel für *gfoldl*:

$$\mathit{gfoldl}\ (\$)\ \mathit{id}\ x = x$$

Weitere Funktionen

Basierend auf der Data Klasse können wir eine Funktion *everywhere* definieren, die eine Funktion auf alle Teile einer Datenstruktur anwendet:

$$everywhere :: Data\ b \Rightarrow (forall\ a\ Data\ a \Rightarrow a \rightarrow a) \rightarrow b \rightarrow b$$
$$everywhere\ f = f \circ gmapT\ (everywhere\ f)$$

Weitere Funktionen

Basierend auf der Data Klasse können wir eine Funktion *everywhere* definieren, die eine Funktion auf alle Teile einer Datenstruktur anwendet:

$$everywhere :: Data\ b \Rightarrow (forall\ a\ Data\ a \Rightarrow a \rightarrow a) \rightarrow b \rightarrow b$$
$$everywhere\ f = f \circ gmapT\ (everywhere\ f)$$

Analog dazu können wir eine Funktion *everything* definieren, die eine Abfrage f auf alle Teile anwendet und mittels eines Kombinators k verknüpft:

$$everything :: Data\ b \Rightarrow (a \rightarrow a \rightarrow a) \rightarrow (forall\ b\ Data\ b \Rightarrow b \rightarrow a) \rightarrow b \rightarrow a$$
$$everything\ k\ f\ x = gfoldl\ k\ (f\ x)\ (gmapQ\ (everything\ k\ f)\ x)$$

Beispiel: gsize

Die Funktion *gsize* soll die Anzahl aller Konstruktoren in einer Datenstruktur berechnen:

$$\textbf{newtype}\ \mathit{IntBox}\ a = \mathit{IntBox}\ \{\mathit{unBox} :: \mathit{Int}\}$$

$$\begin{aligned}
&\mathit{gsize} :: \mathit{Data}\ a \Rightarrow a \to \mathit{Int}\\
&\mathit{gsize} = \mathit{unBox} \circ \mathit{gfoldl}\ k\ (\lambda_ \to \mathit{IntBox}\ 1)\\
&\quad \textbf{where}\ k\ (\mathit{IntBox}\ h)\ t = \mathit{IntBox}\ (h + \mathit{gsize}\ t)
\end{aligned}$$

Beispiel: gsize

Die Funktion *gsize* soll die Anzahl aller Konstruktoren in einer Datenstruktur berechnen:

$$\textbf{newtype}\ \mathit{IntBox}\ a = \mathit{IntBox}\ \{\mathit{unBox} :: \mathit{Int}\}$$

$$\mathit{gsize} :: \mathit{Data}\ a \Rightarrow a \to \mathit{Int}$$
$$\mathit{gsize} = \mathit{unBox} \circ \mathit{gfoldl}\ k\ (\lambda\_ \to \mathit{IntBox}\ 1)$$
$$\quad \textbf{where}\ k\ (\mathit{IntBox}\ h)\ t = \mathit{IntBox}\ (h + \mathit{gsize}\ t)$$

Der *IntBox* type ist nötig um die Funktion *gfoldl* direkt anwenden zu können.
Die Typevariable *a* repräsentiert einen Phantom-Typ, der nur für die Type-Korrektheit nötig ist.

Beispiel: total

Folgende Funktion summiert alle Literale in einem arithmetischen Ausdruck.

$$
\begin{aligned}
&\mathit{total} :: (\mathit{forall}\ a\ \mathit{Data}\ a \Rightarrow a \to \mathit{Int}) \\
&\mathit{total} = \mathit{everything}\ (+)\ (0\ `\mathit{mkQ}`\ \mathit{lit}) \\
&\qquad \textbf{where}\ \mathit{lit} :: \mathit{ArithExpr}\ \mathit{Int} \to \mathit{Int} \\
&\qquad\qquad \mathit{lit}\ (\mathit{Lit}\ x) = x \\
&\qquad\qquad \mathit{lit}\ x \qquad\quad = 0
\end{aligned}
$$

Der Kombinator *mkQ* wandelt eine typ-spezifische Funktion vom Typ $a \to b$ in eine generische Abfrage Funktion um, die 0 als Default-Wert für einen nicht passenden Typ verwendet.

Beispiel: total

Der Kombinator *mkQ* kann wie folgt implementiert werden:

$$
\begin{aligned}
&mkQ :: (Typeable\ a, Typeable\ b) \Rightarrow \\
&\qquad c \to (b \to c) \to a \to c \\
&(r\ `mkQ`\ q)\ a = \textbf{case}\ cast\ a\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Just\ b \quad \to q\ b \\
&\qquad\qquad\qquad\qquad Nothing \to r
\end{aligned}
$$

Beachte: Dieser Code ist Teil `Data.Generic` Bibliothek, die SYB implementiert. Explizite *cast* Operationen werden in der Regel im User-Code nicht auftauchen.

Weitere Bibliotheksfunktionen

```
  -- Compute size of an arbitrary data structure
gsize :: Data a ⇒ a → Int
gsize t = 1 + sum (gmapQ gsize t)

  -- Find (unambiguously) an immediate subterm of a given type
gfindtype :: (Data x, Typeable y) ⇒ x → Maybe y
gfindtype = singleton
    ∘ foldl unJust []
    ∘ gmapQ (Nothing `mkQ` Just)
where
  unJust l (Just x) = x : l
  unJust l Nothing = l
  singleton [s] = Just s
  singleton _ = Nothing
```

Weitere Bibliotheksfunktionen

```
    -- Compute size of an arbitrary data structure
gsize :: Data a ⇒ a → Int
gsize t = 1 + sum (gmapQ gsize t)

    -- Find (unambiguously) an immediate subterm of a given type
gfindtype :: (Data x, Typeable y) ⇒ x → Maybe y
gfindtype = singleton
    ∘ foldl unJust []
    ∘ gmapQ (Nothing `mkQ` Just)
where
    unJust l (Just x) = x : l
    unJust l Nothing = l
    singleton [s] = Just s
    singleton _ = Nothing
```

Ausserdem gibt es für die gängigen higher-order Funktionen auch monadische Varianten.

Erweiterbarkeit generischer Funktionen

Ein Nachteil von SYB ist, dass eine einmal definierte generische Funktion nicht mehr für einen speziellen Datentyp erweitert werden kann.

Dies ist wünschenswert, wenn die Funktionalität vom konkreten Typ abhängen sollen, z.B. um die generische Funktion effizienter zu machen.

Diese Einschränkung kann mit Hilfe von Typ-Klassen umgangen werden. Dabei werden die generischen Funktion auf Klassen-Ebene "ge-lifted" und können dort spezialisiert werden.

Erweiterbarkeit generischer Funktionen: Beispiel

Beispiel: eine erweiterbare Version von *gsize*:

class *Size a* **where** $\{ gsize :: a \rightarrow Int \}$

Der default Fall wird durch eine allgemeine Instanz definiert:

instance *Size a* **where** $\{ gsize\ x = ... \}$

Wir können auch spezialisierte Instanzen, z.B. für Listen definieren:

instance *Size a* $\Rightarrow$ *Size* $[a]$ **where**
gsize x = ...

In diesem Ansatz müssen auch Funktionen wie *gmapQ* auf Typ-Klassen Ebene gelifted werden. Dies erfordert Abstraktionen über Typ-Klassen, die mittels (explizit codierten) Dictionaries realisiert werden können.

Implementierung der Typeable Klasse

Zur Erinnerung, das Interface der *Typeable* Klasse:

```
class Typable a where
  typeOf :: a → TypeRep
```

Die *typeOf* Funktion kann als eindeutige Abbildung von Typen auf deren Repräsentationen wie folgt definiert werden:

```
data TypeRep = TR String [TypeRep]

instance Typeable Int where
  typeOf x = TR "Prelude.Int" []

instance Typeable a ⇒ Typeable [a] where
  typeOf x = TR "Prelude.List" [typeOf (get x)]
               where get :: [a] → a
                     get = ⊥

...
```

Implementierung der Typeable Klasse

Nun kann eine typ-sichere Version eines Typ-casts wie folgt definiert werden:

$$
\begin{aligned}
&\mathit{cast} :: (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow a \to \mathit{Maybe}\ b \\
&\mathit{cast}\ x = r\ \textbf{where}\ r = \textbf{if}\ \mathit{typeOf}\ x \equiv \mathit{typeOf}\ (\mathit{get}\ r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \mathit{Just}\ (\mathit{unsafeCoerce}\ x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad \mathit{get} :: \mathit{Maybe}\ a \to a \\
&\qquad\qquad\qquad\qquad \mathit{get}\ x = \bot
\end{aligned}
$$

Beachte:

- Die **unsichere** Funktion *unsafeCoerce* ist die Identitätsfunktion, und passt den Typ an.
- Die *typeOf* Funktion wertet ihr Argument nie aus. Vom Argument wird nur der Typ benötigt.
- Dies ist high-level System Code, und kein Code den der Benutzer schreiben muss.

Type-indexed data-types

Manchmal ist es wünschenswert einen Datentyp um einen anderen (parametrisierten) Datentyp zu erweitern, z.b. um einen Datentyp mit "Löchern" zu modellieren.

Eine Möglichkeit dies zu realisieren sind **type families**, die als Haskell Erweiterung von GHC unterstützt werden.

Type families ermöglichen es, unter Einschränkungen, benannte **Funktionen auf Typ-Ebene** zu definieren.

Type-indexed data-types

Manchmal ist es wünschenswert einen Datentyp um einen anderen (parametrisierten) Datentyp zu erweitern, z.b. um einen Datentyp mit "Löchern" zu modellieren.

Eine Möglichkeit dies zu realisieren sind **type families**, die als Haskell Erweiterung von GHC unterstützt werden.

Type families ermöglichen es, unter Einschränkungen, benannte **Funktionen auf Typ-Ebene** zu definieren.
Analogie:

Typen	Funktionen
Algeb. Datentypen	Polymorphe Funktionen
Type families	Überladene Funktionen

Beispiel: Datentyp mit Löchern

Wir wollen einen beliebigen Datentyp so erweitern, dass er Löcher, also undefinerte Komponenten, enthalten kann.

Eine naive Implementierung liefert nicht das gewünschte Ergebnis:

$$\textbf{type}\ \mathit{Ext}\ a = \mathit{Unit} : + : a$$

da Löcher, modelliert durch *Unit*, nur auf oberster Ebene des Types verwendet werden können. Wir wollen aber *jedes* Auftreten des Typs derart erweitern.

Type-indexed data-types

Dazu machen wir zunächst Typen explizit:

$$
\begin{aligned}
&\textbf{data } \mathit{Unit} \quad\quad r = \mathit{Unit}\\
&\textbf{data } \mathit{Id} \quad\quad r = \mathit{Id}\ \{\mathit{unId} :: r\}\\
&\textbf{data } \mathit{K}\ a \quad\quad r = \mathit{K}\ a\\
&\textbf{data } \mathit{Sum}\ f\ g\ r = \mathit{Inl}\ (f\ r) \mid \mathit{Inr}\ (g\ r)\\
&\textbf{data } \mathit{Prod}\ f\ g\ r = \mathit{Prod}\ (f\ r)\ (g\ r)
\end{aligned}
$$

Mit diesen Definitionen können wir jeden algebraischen Datentyp in Haskell in einen expliziten Wert abbilden: in eine Summe von Produkten.

Das zusätzliche Typ-Argument r wird dazu verwendet, um Rekursion zu modellieren: der *Id* Typ in den obigen Definitionen deckt den rekursiven Fall ab.

Type-indexed data-types

Um alle regulären Typen zu sammeln, definieren wir nun folgende Typ-Klasse mit einem assozierten Typsynonym:

$$
\begin{array}{l}
\textbf{class } \mathit{Regular}\ a\ \textbf{where} \\
\quad \textbf{type } \mathit{PF}\ a :: * \to * \\
\quad \mathit{from} \quad\quad :: a \to (\mathit{PF}\ a)\ a \\
\quad \mathit{to} \quad\quad\ :: (\mathit{PF}\ a)\ a \to a
\end{array}
$$

Die Funktion *from* bildet die oberste Ebene eines Typs a in unsere explizite Darstellung der Typen ab. Der “pattern functor” ist über das rekursive Argument parametrisiert.

Type-indexed data-types

Beispiel: Listen

```
instance Regular [a] where
  type PF [a] = Sum Unit (Prod (K a) Id)
  from []       = Inl Unit
  from (x : xs) = Inr (Prod (K x) (Id xs))
  to (Inl Unit)                  = []
  to (Inr (Prod (K x) (Id xs))) = x : xs
```

Type-indexed data-types

Beispiel: Listen

```
instance Regular [a] where
  type PF [a] = Sum Unit (Prod (K a) Id)
  from []       = Inl Unit
  from (x : xs) = Inr (Prod (K x) (Id xs))
  to (Inl Unit)                    = []
  to (Inr (Prod (K x) (Id xs)))    = x : xs
```

Beachte: Das Typsynonym *PF* entspricht, auf Typ-Ebene, einer überladenen Funktion: die Definition ist abhängig vom instanzierten Typ.

Type-indexed data-types

Nun können wir einen Datentyp mit Löchern wie folgt, mittels eines Fixpunkt-Iterators auf der expliziten Repräsentation von Typen definieren:

$$\textbf{newtype}\ \mathit{Fix}\ f = \mathit{In}\ \{\mathit{out} :: f\ (\mathit{Fix}\ f)\}$$
$$\textbf{type}\ \mathit{Ext}\ f = \mathit{Sum}\ \mathit{Unit}\ f$$
$$\textbf{type}\ \mathit{Hole}\ f = \mathit{Fix}\ (\mathit{Ext}\ (\mathit{PF}\ f))$$

Type-indexed data-types

Wir können nun eine monadische Funktion definieren, die einen expliziten Datentyp traversiert und f auf die Komponenten anwendet:

```
class TraverseM f where
  traverseM :: Monad m ⇒ (a → m b) → f a → m (f b)

instance TraverseM Unit where
  traverseM Unit = return Unit

instance TraverseM Id where
  traverseM f (Id x) = liftM Id (f x)

instance traverseM (K a) where
  traverseM _ (K a) = return (K a)

instance (TraverseM f, TraverseM g) ⇒ TraverseM (Sum f g) where
  traverseM f (Inl x) = liftM Inl (traverseM f x)
  traverseM f (Inr x) = liftM Inr (traverseM f x)

instance (TraverseM f, TraverseM g) ⇒ TraverseM (Prod f g) where
  traverseM f (Prod x y) = liftM2 Prod (traverseM f x) (traverseM f y)
```

Type-indexed data-types

Mit dieser Funktion können wir nun unsere generische *ginsert* Funktion definieren:

```
ginsert' :: Insert a ⇒ Hole a → State [a] (Maybe a)
ginsert' (In (Inl Unit)) = do    -- if it's a hole, fill it
  l ← get
  case l of
       []      → return Nothing
       (x : t) → put t ≫ return (Just x)
ginsert' (In (Inr x)) = do       -- if it's proper data, traverse it
  t ← traverseM ginsert' x
  return (traverseM id t ≫= return ∘ to)
```

Type-indexed data-types

$$ginsert :: Insert\ a \Rightarrow [a] \rightarrow Hole\ a \rightarrow Maybe\ a$$
$$ginsert\ as\ h = evalState\ (ginsert'\ h)\ as$$

Die *Insert* Klasse fordert lediglich, dass ihre Instanzen auch Instanzen der Klassen *Regular* und *TraverseM* sind:

$$\textbf{class}\ (Regular\ a, TraverseM\ (PF\ a)) \Rightarrow Insert\ a$$

Type-indexed data-types

Für unser Beispiel arithmetischer Ausdrücke kann der Compiler nun automatisch Instanzen erzeugen:

```
instance Regular (ArithExpr a) where
type PF (ArithExpr a) = Sum (K String)
   (Sum (K a)
   (Sum (Prod Id Id)
   (Sum (Prod Id Id)
   (Sum (Prod Id Id)
   (Sum (Prod Id Id))))))
from (Var s) = Inl (K s)
 ...
to (Inl (K s)) = Var s
 ...
instance Insert (ArithExpr Int)
```

Type-indexed data-types

Ein Beispiel der Anwendung unserer *ginsert* Funktion, auf einen Datentyp arithmetischer Ausdrücke mit "Löchern":

```
expr :: Hole (ArithExpr Int)
expr = sum hole (sum (lit 2) hole)   -- encodes: . + (2 + .)
       where value x   = In (Inr x)
             hole      = In (Inl Unit)
             sum a b   = value (Inr (Inr (Inl (Prod (Id a) (Id b)))))
             lit n     = value (Inr (Inl (K n)))

ins1, ins2 :: Maybe (ArithExpr Int)
ins1 = ginsert [Lit 4, Lit 6] expr
ins2 = ginsert [Lit 4] expr
```

Auswertung: *ins1* $\rightsquigarrow$ Just (Lit 4 + (Lit 2 + Lit 6))
Auswertung: *ins2* $\rightsquigarrow$ Nothing
Den "boilerplate" in der expliziten Repräsentation von Typen kann man noch weiter verringern.

Zusammenfassung

(Datentyp-) Generische Programme abstrahieren über die Struktur eines Datentyps.

Damit wird es möglich Funktionen wie *map* über beliebige Datenstrukturen zu definieren.

Scrap Your Boilerplate (SYB) ist ein in GHC unterstützter Ansatz zur generischen Programmierung, der mittels des **deriving** Konstrukts und vordefinierter Primitive eine einfache Formulierung generischer Programme ermöglicht.

Im Gegensatz zu anderen Ansätzen werden Typen nicht als explizite Struktur dem Programmierer zur Verfügung gestellt.

Literatur

- J. Jeurnig, S. Leather, J.P. Magalhaes, A.R. Yakushev. "Libraries for Generic Programming in Haskell", Summer School on Advanced Functional Programming, 2008.
- R. Lämmel, S. Peyton Jones. "Scrap Your Boilerplate". In *TLDI'03 — Types in Language Design and Implementation*, 2003.
- J. Gibbons. "Datatype-generic Programming", LNCS 4719, pp 1–71, 2007.