Generische Record-Kombinatoren mit statischer Typprüfung

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau

Lehrstuhlkolloquium am 13. Januar 2010

Überblick

Records in Haskell

- Beispiel-Record:

$$\begin{aligned} \mathit{employee} = \mathit{Employee}\ \{ & \\ \quad \mathit{surname} &= \texttt{"Jeltsch"}, \\ \quad \mathit{age} &= 31, \\ \quad \mathit{room} &= \texttt{"EH/202"} \\ \} & \end{aligned}$$

- benötigte Typdeklaration:

$$\begin{aligned} \textbf{data}\ \mathit{Employee} = \mathit{Employee}\ \{ & \\ \quad \mathit{surname} &:: \mathit{String}, \\ \quad \mathit{age} &:: \mathit{Int}, \\ \quad \mathit{room} &:: \mathit{String} \\ \} & \end{aligned}$$

Record-Operationen

- Selektion mittels Musteranpassung:

$$\begin{aligned}
&\mathit{roomInfo} :: \mathit{Employee} \to \mathit{String}\\
&\mathit{roomInfo}\ (\mathit{Employee}\ \{\\
&\qquad\qquad \mathit{room} \quad\ = \mathit{place},\\
&\qquad\qquad \mathit{surname} = \mathit{name}\\
&\qquad\quad \})\qquad\qquad\qquad\qquad = \mathit{info}\ \textbf{where}\\
&\quad \mathit{info} = \mathit{name} \mathbin{++} \texttt{"works in"}\ \mathit{place} \mathbin{++} \texttt{"."}
\end{aligned}$$

- Ändern von Feldwerten:

$$\mathit{employee}\ \{\mathit{room} = \texttt{"HG/2.14"}, \mathit{age} = 33\}$$

- Möglichkeiten für Muster und Änderungsausdrücke:
  - andere Reihenfolge der Felder
  - Weglassen von Feldern

Modifizieren statt Überschreiben

- Record von Änderungen:

$$\begin{aligned} \mathit{employeeMod} = \mathit{EmployeeMod}\ \{ & \\ \mathit{age} \quad &= (+2), \\ \mathit{room} &= \mathit{const}\ \texttt{"HG/2.14"} \\ \} & \end{aligned}$$

- Typ des Modifikations-Records:

$$\begin{aligned} \textbf{data}\ \mathit{EmployeeMod} = \mathit{EmployeeMod}\ \{ & \\ \mathit{age} \quad &:: \mathit{Int} \quad \to \mathit{Int}, \\ \mathit{room} &:: \mathit{String} \to \mathit{String} \\ \} & \end{aligned}$$

- gewünscht: Funktion *modify* zum Durchführen der Modifikation:

$$\mathit{modify}\ \mathit{employeeMod}\ \mathit{employee}$$

Generische Record-Kombinatoren

- *modify* soll mit allen zueinander passenden Record-Typen funktionieren
- dazu notwendig:
  - Funktionen, die mit unterschiedlichen Record-Typen arbeiten, müssen definierbar sein
  - Beziehungen zwischen Record-Typen müssen in Funktionstypen ausgedrückt werden können
- in diesem Vortrag: ein Record-System, was das ermöglicht
- keine Ad-Hoc-Lösung:
  - generische Basis zum Erstellen verschiedener Record-Kombinatoren
  - *modify* als Spezialfall

Überblick

Heterogene Listen

- induktive Definition von Wertmengen mittels algebraischer Datentypen
- Listen als Beispiel:

$$\textbf{data}\ \mathit{List}\ \mathit{el} = \mathit{Nil} \mid \mathit{Snoc}\ (\mathit{List}\ \mathit{el})\ \mathit{el}$$

- induktive Definition von Typmengen:
  - verschiedene Typen für verschiedene Alternativen
  - Typparameter stellen Parameter der Alternative dar
- heterogene Listen (Tupel):

$$\textbf{data}\ X = X$$

$$\textbf{data}\ \mathit{init} :\& \mathit{last} = \mathit{init} :\& \mathit{last}$$

- Beispielliste:

$$X :\& \texttt{"Jeltsch"} :\& 31 :\& \texttt{"EH/202"}$$

- Typ dieser Liste:

$$X :\& \mathit{String} :\& \mathit{Int} :\& \mathit{String}$$

Records als spezielle Listen

- Record ist heterogene Liste von Feldern

  Feld Paar aus Name und Wert

  Feld-Typ Paar aus Name und Wertetyp
- Konsequenz: Name muss sowohl auf Werte- als auch auf Typebene darstellbar sein
- Repräsentation durch Typ- und Datenkonstruktor:

  data *Name* = *Name*
- Feldtyp ist lediglich Paartyp (mit schönerer Notation):

  data *name* ::: *val* = *name* := *val*
- Klasse aller Record-Typen induktiv definiert:

  class *Record rec*

  instance *Record X*

  instance (*Record rec*) ⇒
  Record (*rec* :& *name* ::: *val*)

Beispiel-Record

- Deklaration der Feldnamen:

$$\begin{aligned}&\textbf{data}\ \mathit{Surname} = \mathit{Surname}\\&\textbf{data}\ \mathit{Age} \quad\ = \mathit{Age}\\&\textbf{data}\ \mathit{Room} \quad = \mathit{Room}\end{aligned}$$

- Record:

$$\begin{aligned}X\ :\&\ \mathit{Surname} &:= \texttt{"Jeltsch"}\\ :\&\ \mathit{Age} &:= 31\\ :\&\ \mathit{Room} &:= \texttt{"EH/202"}\end{aligned}$$

- Typ des Records:

$$\begin{aligned}X\ :\&\ \mathit{Surname} &::: \mathit{String}\\ :\&\ \mathit{Age} &::: \mathit{Int}\\ :\&\ \mathit{Room} &::: \mathit{String}\end{aligned}$$

Konvertieren von Records

- erstebenswerte Features:
  - Reihenfolge von Record-Feldern egal
  - Ignorieren uninteressanter Felder möglich
- Realisierung mittels Record-Konvertierung:
  - Ändern der Reihenfolge der Felder
  - Entfernen von Feldern
- Klasse von Paaren aus Quell- und Zieltyp mit Konvertierungsmethode:

  class *Convertible rec rec′* **where**

  convert :: *rec* → *rec′*

- Musteranpassung auf Basis von Record-Konvertierung:
  - Funktionen geben statt Records ρ entsprechende Records *convert* ρ zurück
  - Muster fixiert durch Auflistung der Feldnamen den Zieltyp der Konvertierung

Überblick

Die Typen der Argumente von *modify*

- Typ des Modifikations-Records verglichen mit Typ des Daten-Records:
  - Feldtypen $v \to v$ statt v
  - eventuell andere Reihenfolge der Felder
  - eventuell Weglassen von Feldern
- Behandlung der letzten beiden Punkte mittels Record-Konvertierung
- im Folgenden nur Behandlung des ersten Punktes
- eingeschränkte *modify*-Funktion:
  - gleiche Feldreihenfolge
  - kein Weglassen von Feldern

Familien verwandter Record-Typen

- Record-Typ zusammengesetzt aus zwei Komponenten:

  Schema Liste von Paaren aus je einem Namen und einem Typ, Sorte genannt:

  $$X :\& name_1 ::: sort_1 :\& \ldots :\& name_n ::: sort_n$$

  Stil Funktion über Typen
- Wertetypen des Record-Typs entstehen durch Anwenden des Stils auf die Sorten
- verwandte Record-Typen besitzen selbes Schema, aber unterschiedliche Stile
- Deklaration von Schematypen:

  $$\begin{aligned} &\textbf{data}\ X && style = X \\ &\textbf{data}\ (rec :\& field) && style = rec\ style :\& field\ style \\ &\textbf{data}\ (name ::: sort) && style = name := style\ sort \end{aligned}$$

- *Record* jetzt Klasse aller Record-Schemata

Der Typ von *modify*

- Wertetypen des Daten-Records werden als Sorten benutzt
- zwei Record-Stile:

  einfach $\lambda val \to val$

  Änderungen $\lambda val \to (val \to val)$
- Typ von *modify*:

$$(Record\ rec) \Rightarrow rec\ (\lambda val \to (val \to val)) \to rec\ (\lambda val \to val) \to rec\ (\lambda val \to val)$$

- Problem: keine λ-Ausdrücke auf der Typebene
- Lösung durch Defunktionalisierung
- benötigt Typfamilien

Typen in Standard-Haskell

- algebraische Datentypen:

  $$\textbf{data}\ \mathit{List}\ \mathit{el} = \mathit{Nil} \mid \mathit{Snoc}\ (\mathit{List}\ \mathit{el})\ \mathit{el}$$

- Typsynonyme:

  $$\textbf{type}\ \mathit{Endofunction}\ \mathit{val} = \mathit{val} \to \mathit{val}$$

- gemeinsame Eigenschaft: Implementierung ist unabhängig von Typparametern

Typfamilien

- mit Typfamilien verschiedene Implementierungen für verschiedene Parameter möglich
- Datentypfamilien:

  data family $Map\ key\ \ \ val$

  data instance $Map\ Bool\ \ \ val =$
  $BoolMap\ val\ val$

  data instance $Map\ (key_1, key_2)\ val =$
  $PairMap\ (Map\ key_1\ (Map\ key_2\ val))$

- Typsynonymfamilien:

  type family $Element\ set$

  type instance $Element\ (Set\ el)\ = el$

  type instance $Element\ BitArray = Int$

Defunktionalisierung auf der Typebene

- Repräsentieren von Typfunktionen durch (leere) Typen
- Typsynonymfamilie, die Funktionsanwendung beschreibt:

  type family *App funRep arg*

- Repräsentieren der Typfunktion $\lambda\alpha \rightarrow \tau$:

  data *Λ*

  type instance *App Λ* $\alpha = \tau$

- geänderte Deklaration des Typs der Record-Felder:

  data (*name* ::: *sort*) *style* =
  name := *App style sort*

Der Typ von *modify* mit Defunktionalisierung

- Repräsentieren der beiden Record-Stile:

  data *PlainStyle*

  data *ModStyle*

  type instance *App PlainStyle val* = *val*

  type instance *App ModStyle val* = *val* → *val*

- der Typ von *modify*:

$$(\mathit{Record}\ \mathit{rec}) \Rightarrow \mathit{rec}\ \mathit{ModStyle} \rightarrow \mathit{rec}\ \mathit{PlainStyle} \rightarrow \mathit{rec}\ \mathit{PlainStyle}$$

Überblick

Induktion über Record-Schemata

- Implementierung von *modify* als *Record*-Methode:

$$\begin{aligned}
&\textbf{class } \mathit{Record}\ \mathit{rec}\ \textbf{where}\\
&\quad \mathit{modify} :: \mathit{rec}\ \mathit{ModStyle} \to\\
&\qquad\qquad \mathit{rec}\ \mathit{PlainStyle} \to\\
&\qquad\qquad \mathit{rec}\ \mathit{PlainStyle}\\
&\textbf{instance } \mathit{Record}\ X\ \textbf{where}\\
&\quad \mathit{modify}\ X\ X = X\\
&\textbf{instance } (\mathit{Record}\ \mathit{rec}) \Rightarrow\\
&\qquad\qquad \mathit{Record}\ (\mathit{rec}\ :\&\ \mathit{name} ::: \mathit{val})\ \textbf{where}\\
&\quad \mathit{modify}\ (\mathit{mods}\ :\&\ \mathit{name} := \mathit{mod})\\
&\qquad\qquad (\mathit{rec}\ \ :\&\ _\ \ := \mathit{val}) \ = \mathit{rec}'\ \textbf{where}\\
&\qquad \mathit{rec}' = \mathit{modify}\ \mathit{mods}\ \mathit{rec}\ :\&\ \mathit{name} := \mathit{mod}\ \mathit{val}
\end{aligned}$$

- Problem: Menge von *Record*-Methoden fixiert
- Lösung: Faltungsfunktion für Record-Schemata

Faltung

- Faltungsfunktion kapselt Induktionsprinzip
- Faltungsfunktion für Listen:

$$\begin{aligned}
&\mathit{fold} :: \mathit{accu} \to (\mathit{accu} \to \mathit{el} \to \mathit{accu}) \to \mathit{List}\ \mathit{el} \to \mathit{accu}\\
&\mathit{fold}\ \mathit{nilAlt}\ _\ \ \mathit{Nil} = \mathit{nilAlt}\\
&\mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt}\ (\mathit{Snoc}\ \mathit{els}\ \mathit{el}) = \mathit{accu}\ \textbf{where}\\
&\quad \mathit{accu} = \mathit{snocAlt}\ (\mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt}\ \mathit{els})\ \mathit{el}
\end{aligned}$$

- Implementierung einer Funktion $\mathit{sum} :: \mathit{List}\ \mathit{Int} \to \mathit{Int}$:
  - ohne *fold*:

$$\begin{aligned}
&\mathit{sum}\ \mathit{Nil} = 0\\
&\mathit{sum}\ (\mathit{Snoc}\ \mathit{nums}\ \mathit{num}) = \mathit{sum}\ \mathit{nums} + \mathit{num}
\end{aligned}$$

  - mit *fold*:

$$\mathit{sum} = \mathit{fold}\ 0\ (+)$$

Faltungsfunktion für Record-Schemata

```
class Record rec where
  fold :: thing X                                          →
          (∀rec name sort.(Record rec) ⇒
           thing rec → thing (rec :& name ::: sort)) →
          thing rec
instance Record X where
  fold nilAlt _ = nilAlt
instance (Record rec) ⇒
           Record (rec :& name ::: sort) where
  fold nilAlt snocAlt = snocAlt (fold nilAlt snocAlt)
```

Implementierung von *modify* mittels *fold*

- *thing* entspricht dem Typ von *modify*:

$$\begin{aligned}&\textbf{newtype}\ \mathit{ModThing}\ \mathit{rec} =\\ &\quad \mathit{ModThing}\ (\mathit{rec}\ \mathit{ModStyle} \to\\ &\qquad\qquad \mathit{rec}\ \mathit{PlainStyle} \to\\ &\qquad\qquad \mathit{rec}\ \mathit{PlainStyle})\end{aligned}$$

- *modify* entpackt lediglich das Ergebnis von *fold*:

$$\begin{aligned}&\mathit{modify}\ \mathit{modRec}\ \mathit{rec} = \mathit{result}\ \textbf{where}\\ &\quad \mathit{ModThing}\ \mathit{result} = \mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt}\end{aligned}$$

Implementierung der Alternativen

```
nilAlt :: ModThing X
nilAlt = ModThing nilModify where
  nilModify X X = X
snocAlt :: (Record rec) ⇒
           ModThing rec                          →
           ModThing (rec :& name ::: sort)
snocAlt (ModThing modify) = ModThing snocModify where
  snocModify (mods :& name := mod)
             (rec  :& _    := val)  = accu where
    accu = modify mods rec :& name := mod val
```

Wirklich eine Faltung?

- Vergleich mit Faltungsfunktion für Listen
- letzte Elemente und komplette Liste tauchen als Funktionsargumente auf:

$$\mathit{thing} \to (\mathit{thing} \to \mathit{el} \to \mathit{thing}) \to \mathit{List}\ \mathit{el} \to \mathit{thing}$$

- Analogien zwischen Listen-Faltung und Record-Schema-Faltung:

  letztes Element $\leftrightarrow$ Name und Sorte des letzten Feldes

  komplette Liste $\leftrightarrow$ komplettes Record-Schema

- Name und Sorte des letzten Feldes sowie komplettes Record-Schema tauchen nicht als Funktionsargumente auf

Natürlich!

- Anwenden von Äquivalenzen auf den Typ von *fold*:
  - von Allquantifizierung zu abhängigen Typen:

$$\forall \alpha :: \kappa.\tau \cong (\alpha :: \kappa) \to \tau$$

  - Absenken einer Allquantifizierung:

$$\forall \alpha :: \kappa.\tau \to \tau' \cong \tau \to \forall \alpha :: \kappa.\tau', \text{ falls } \alpha \notin \mathit{FV}(\tau)$$

- Resultat der Umformung:

$$\begin{aligned}
&\mathit{thing}\ X \\
\to\ &(\forall \mathit{rec}.(\mathit{Record}\ \mathit{rec}) \Rightarrow \mathit{thing}\ \mathit{rec} && \to \\
&\qquad (\mathit{name} :: *) && \to \\
&\qquad (\mathit{sort} :: *) && \to \\
&\qquad \mathit{thing}\ (\mathit{rec} :\& \mathit{name} ::: \mathit{sort})) \\
\to\ &(\mathit{rec} :: * \to *) \\
\to\ &\mathit{thing}\ \mathit{rec}
\end{aligned}$$

Überblick

Zusammenfassung und Ausblick

- Zusammenfassung:
  - Record-System als Bibliothek
  - alle Features von Haskells Standard-Record-System unterstützt
  - mittels Record-Schema-Konzept Beziehungen zwischen Record-Typen durch Kombinator-Typen darstellbar
  - beliebige induktive Definitionen über Record-Schemata möglich
- Ausblick:
  - existierende Lösung für Records mit Einschränkungen der Wertetypen
  - Effizienzuntersuchungen wichtig:
    - Record-Konvertierung hat quadratischen Zeitaufwand
    - inwieweit Berechnung zur Compilierzeit?
  - implizite Definition von Namenstypen wünschenswert

Generische Record-Kombinatoren mit statischer Typprüfung

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau

Lehrstuhlkolloquium am 13. Januar 2010