Record Type Families: A Key to Generic Record Combinators

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 20, 2011

Overview

Overview

A typical system of extensible records

- records map names to values:

$$\mathit{wolfgang} = \{\mathit{surname} = \texttt{"Jeltsch"}, \\ \mathit{age} = 33, \\ \mathit{place} = \texttt{"Cottbus"}\}$$

- types of records map names to types:

$$\mathit{wolfgang} :: \{\mathit{surname} :: \mathit{String}, \\ \mathit{age} :: \mathit{Integer}, \\ \mathit{place} :: \mathit{String}\}$$

- only field-related operations:
  - selection
  - modification
  - addition
  - removal
- no support for combinators, i.e., functions that work with complete records

An example combinator

- record of modifications:

$$\mathit{mods} = \{\,\mathit{surname} = \mathit{id},\ \mathit{age} = (+1),\ \mathit{place} = \mathit{const}\ \texttt{"Tallinn"}\,\}$$

- type of the modification record:

$$\mathit{mods} :: \{\,\mathit{surname} :: \mathit{String} \to \mathit{String},\ \mathit{age} :: \mathit{Integer} \to \mathit{Integer},\ \mathit{place} :: \mathit{String} \to \mathit{String}\,\}$$

- function *modify* that performs the modification:

$$\mathit{modify}\ \mathit{mods}\ \mathit{wolfgang} = \{\,\mathit{surname} = \texttt{"Jeltsch"},\ \mathit{age} = 34,\ \mathit{place} = \texttt{"Tallinn"}\,\}$$

Generic record combinators

- *modify* shall work with all modification and data records whose types match:
  - *modify* must be generic
  - type of *modify* must be able to express necessary relationships between the argument types
- *modify* works with complete records
- topic of this talk:

  a record system that allows us to define combinators like *modify*
- implemented as a Haskell library:
  - works with standard GHC
  - key to success are advanced type system features

Overview

Heterogeneous lists

- types for building heterogeneous lists:
  - the empty list:

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

  - non-empty lists, each consisting of an initial list and a last element:

    $$\textbf{data } \delta :\& \ \varepsilon = \delta :\& \ \varepsilon$$

- example list:

$$X :\& \texttt{"Jeltsch"} :\& 33 :\& \texttt{"Cottbus"}$$

- type of this list:

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

Records

- record is heterogeneous list of fields:

  field a pair of a name and a value

  field type a pair of a name and a type

- names appear at the value level and at the type level
- represent names by a type and a data constructor:

  $$\mathbf{data}\ N = N$$

- type of fields:

  $$\mathbf{data}\ \nu ::: \alpha = \nu := \alpha$$

The example data record

- field names:

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

- data record:

$$\begin{aligned}
\mathit{wolfgang} = X\ &{:}\&\ \mathit{Surname} := \texttt{"Jeltsch"} \\
&{:}\&\ \mathit{Age} := 33 \\
&{:}\&\ \mathit{Place} := \texttt{"Cottbus"}
\end{aligned}$$

- type of the data record:

$$\begin{aligned}
\mathit{wolfgang} :: X\ &{:}\&\ \mathit{Surname} ::: \mathit{String} \\
&{:}\&\ \mathit{Age} ::: \mathit{Integer} \\
&{:}\&\ \mathit{Place} ::: \mathit{String}
\end{aligned}$$

Overview

Record type families

- allow us to specify relationships between record types
- record type now built from two ingredients:

  scheme
  : a list of pairs, each consisting of a name and a so-called sort:

  $$X :\& \nu_1 ::: \varsigma_1 :\& \ldots :\& \nu_n ::: \varsigma_n$$

  style
  : a type-level function σ

- types of field values are generated on the fly by applying the style to the sorts:

  $$\sigma\ \varsigma_1, \ldots, \sigma\ \varsigma_n$$

- families of related record types can be generated by combining the same scheme with different styles

Implementation

- record scheme is a type with a sort parameter
- type $\rho\ \sigma$ is the record type with scheme ρ and sort σ
- type declarations:

$$
\begin{aligned}
&\textbf{data}\ X && \sigma = X \\
&\textbf{data}\ (\rho :\&\ \varphi) && \sigma = \rho\ \sigma :\&\ \varphi\ \sigma \\
&\textbf{data}\ (\nu ::: \varsigma) && \sigma = \nu := \sigma\ \varsigma
\end{aligned}
$$

- class *Record* of all record schemes:

$$
\begin{aligned}
&\textbf{class} && \mathit{Record}\ \rho \\
&\textbf{instance} && \mathit{Record}\ X \\
&\textbf{instance}\ (\mathit{Record}\ \rho) \Rightarrow && \mathit{Record}\ (\rho :\&\ \nu ::: \varsigma)
\end{aligned}
$$

The type of *modify*

- record styles:

data	$\lambda\alpha \to \alpha$
modification	$\lambda\alpha \to (\alpha \to \alpha)$

- type of *modify*:

$$\begin{aligned}(\mathit{Record}\ \rho) \Rightarrow\ & \rho\ (\lambda\alpha \to (\alpha \to \alpha)) \to \\ & \rho\ (\lambda\alpha \to \alpha) \to \\ & \rho\ (\lambda\alpha \to \alpha)\end{aligned}$$

- problem:

  no λ-expressions at the type level

- solution:

  defunctionalization at the type level

Defunctionalization at the type level

- type-level functions represented by (empty) types
- type synonym family that describes function application:

  type family *App* φ α

- representation of a type-level function $\lambda\alpha \to \tau$ (where α may occur free in τ):

  data *Λ*

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

- modified declaration of the type of record fields:

  data $(\nu ::: \varsigma)\ \sigma = \nu := \mathit{App}\ \sigma\ \varsigma$

The type of *modify* with defunctionalization

- representations of the two record styles:

  $$\textbf{data}\ \Sigma_{\mathit{Plain}}$$
  $$\textbf{data}\ \Sigma_{\mathit{Mod}}$$
  $$\textbf{type instance}\ \mathit{App}\ \Sigma_{\mathit{Plain}}\ \alpha = \alpha$$
  $$\textbf{type instance}\ \mathit{App}\ \Sigma_{\mathit{Mod}}\ \alpha = \alpha \to \alpha$$

- type of *modify*:

  $$(\mathit{Record}\ \rho) \Rightarrow \rho\ \Sigma_{\mathit{Mod}} \to \rho\ \Sigma_{\mathit{Plain}} \to \rho\ \Sigma_{\mathit{Plain}}$$

Overview

Implementation of *modify*

- make *modify* a method of the *Record* class:

  $$\textbf{class}\ \mathit{Record}\ \rho\ \textbf{where}$$
  $$\quad \mathit{modify} :: \rho\ \Sigma_{\mathit{Mod}} \to \rho\ \Sigma_{\mathit{Plain}} \to \rho\ \Sigma_{\mathit{Plain}}$$

- implement *modify* within the instance declarations of *Record*:

  $$\textbf{instance}\ \mathit{Record}\ X\ \textbf{where}$$
  $$\quad \mathit{modify}\ X\ X = X$$
  $$\textbf{instance}\ (\mathit{Record}\ \rho) \Rightarrow \mathit{Record}\ (\rho :\&\ \nu ::: \alpha)\ \textbf{where}$$
  $$\quad \mathit{modify}\ (q :\&\ \_ := f)\ (r :\&\ \nu := x) = \mathit{modify}\ q\ r :\&\ \nu := f\ x$$

- definition of *modify* uses induction over record schemes
- problem:

  impossible to add further methods later

A fold combinator for record schemes

- induction principles are captured by fold combinators
- all inductive definitions on record schemes expressible as applications of a record scheme fold operator
- implement such a combinator:

class *Record* ρ **where**

$$\begin{aligned}
&\mathit{fold} :: \theta\ X \rightarrow \\
&\quad (\forall \rho\ \nu\ \varsigma.(\mathit{Record}\ \rho) \Rightarrow \\
&\quad\quad \theta\ \rho \rightarrow \theta\ (\rho :\&\ \nu ::: \varsigma)) \rightarrow \\
&\quad \theta\ \rho
\end{aligned}$$

instance *Record* X **where**

$$\mathit{fold}\ f_X\ \_ = f_X$$

instance $(\mathit{Record}\ \rho) \Rightarrow \mathit{Record}\ (\rho :\&\ \nu ::: \varsigma)$ **where**

$$\mathit{fold}\ f_X\ f_{(:\&)} = f_{(:\&)}\ \left(\mathit{fold}\ f_X\ f_{(:\&)}\right)$$

Implementation of *modify* using *fold*

- replacement type for the θ-variable:

$$\textbf{type}\ \Theta_{modify}\ \rho = \rho\ \Sigma_{Mod} \to \rho\ \Sigma_{Plain} \to \rho\ \Sigma_{Plain}$$

- implementation of *modify*:

$$\begin{aligned}
&modify :: (Record\ \rho) \Rightarrow \\
&\qquad \rho\ \Sigma_{Mod} \to \rho\ \Sigma_{Plain} \to \rho\ \Sigma_{Plain} \\
&modify = fold\ f_X\ f_{(:\&)}\ \textbf{where} \\
&\quad f_X :: \Theta_{modify}\ X \\
&\quad f_X\ X\ X = X \\
&\quad f_{(:\&)} :: (Record\ \rho) \Rightarrow \\
&\qquad \Theta_{modify}\ \rho \to \Theta_{modify}\ (\rho :\&\ \nu ::: \varsigma) \\
&\quad f_{(:\&)}\ g = \lambda(q :\&\ \nu := f) \\
&\qquad\qquad (r :\&\ _ := x) = g\ q\ r :\&\ \nu := f\ x
\end{aligned}$$

- cheated a bit:

Θ_{modify} must be a proper type, not a type synonym

Is it really a fold?

- compare the record fold combinator to a fold combinator for lists
- heads of non-empty lists and complete list show up as function arguments:

$$\theta \to (\alpha \to \theta \to \theta) \to [\alpha] \to \theta$$

- analogies between both folds:

$$\begin{aligned} \text{head} &\iff \text{name and sort of last field} \\ \text{complete list} &\iff \text{complete record scheme} \end{aligned}$$

- last name, last sort, and complete record scheme do not show up as arguments:

$$\theta\ X \to (\forall \rho\ \nu\ \varsigma.(\mathit{Record}\ \rho) \Rightarrow \theta\ \rho \to \theta\ (\rho :\&\ \nu ::: \varsigma)) \to \theta\ \rho$$

- they cannot, since they are not values

Yes, it is!

- applying equivalences to the type of *fold*:

$$(\forall \alpha :: \xi . \tau) \cong ((\alpha :: \xi) \to \tau)$$
$$(\forall \alpha :: \xi . \tau \to \tau') \cong (\tau \to \forall \alpha :: \xi . \tau') \text{ if } \alpha \notin \mathrm{FV}(\tau)$$

- original type with explicit global quantification of ρ (where Ξ_{Record} denotes "the kind of all records"):

$$\begin{array}{l} \forall (\rho :: \Xi_{Record}). \\ \theta\ X \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\ (\forall \rho\ \nu\ \varsigma . (Record\ \rho) \Rightarrow \theta\ \rho \to \theta\ (\rho :\&\ \nu ::: \varsigma)) \to \\ \theta\ \rho \end{array}$$

- transformation result contains the last name, the last sort, and the complete record scheme as arguments:

$$\begin{array}{l} \theta\ X \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\ (\forall \rho . (Record\ \rho) \Rightarrow \\ \qquad \theta\ \rho \to (\nu :: *) \to (\varsigma :: *) \to \theta\ (\rho :\&\ \nu ::: \varsigma)) \to \\ (\rho :: \Xi_{Record}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\ \theta\ \rho \end{array}$$

Record Type Families: A Key to Generic Record Combinators

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 20, 2011