Generic Record Combinators with Static Type Checking

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

12th International ACM SIGPLAN Symposium
on Principles and Practice of Declarative Programming

July 26–28, 2010

The problem

- records map names to values:

$$\{\mathit{surname} = \texttt{"Jeltsch"}, \mathit{age} = 31, \mathit{room} = \texttt{"EH/202"}\}$$

- types of records map names to types:

$$\{\mathit{surname} :: \mathit{String}, \mathit{age} :: \mathit{Int}, \mathit{room} :: \mathit{String}\}$$

- typically only field-related operations:
  - selection
  - modification
  - addition
  - removal
- wanted:

  combinators, which work on complete records

An example combinator

- record of modifications:

$$\begin{aligned}\{&\mathit{surname} = \mathit{id},\\ &\mathit{age} \quad\;\; = (+2),\\ &\mathit{room} \quad\;\; = \mathit{const}\ \texttt{"HG/2.39"}\}\end{aligned}$$

- type of the modification record:

$$\begin{aligned}\{&\mathit{surname} :: \mathit{String} \to \mathit{String},\\ &\mathit{age} \quad\;\; :: \mathit{Int} \quad\;\; \to \mathit{Int},\\ &\mathit{room} \quad\;\; :: \mathit{String} \to \mathit{String}\}\end{aligned}$$

- function *modify* that performs the modification:
  - works on complete records
  - can modify records of arbitrary structure
  - requires modification records to match the respective data records

A new record system

- supports the definition of generic record combinators
- combinator types can express relationships between record types:

  static checking of relationships
- implemented as a Haskell library

Record type families

- record type built from two ingredients:

  scheme a mapping from names to so-called sorts

  style a type-level function

- record maps each name ν of the scheme to a value of type $\sigma\ \varsigma_\nu$ where
  - σ is the style of the record
  - ς_ν is the sort that the scheme assigns to ν
- families of related record types:
  - same scheme
  - different styles

Representation in Haskell

- field names represented by type and data constructors:

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

- declaration of record scheme types:

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

- example record scheme:

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

- class *Record* of all record schemes

The type of *modify*

- record styles:

$$\begin{aligned} \text{data} &\;\; \lambda val \longrightarrow val \\ \text{modification} &\;\; \lambda val \longrightarrow (val \longrightarrow val) \end{aligned}$$

- type of *modify*:

$$\begin{aligned} (Record\ rec) \Rightarrow\ & rec\ (\lambda val \longrightarrow (val \longrightarrow val)) \longrightarrow \\ & rec\ (\lambda val \longrightarrow val) \longrightarrow \\ & rec\ (\lambda val \longrightarrow val) \end{aligned}$$

- problem:

  no λ-expressions at the type level

- solution:

  defunctionalization at the type level

Implementation of *modify* as a class method

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

$$\begin{aligned}
&\textbf{class } \mathit{Record}\ \mathit{rec}\ \textbf{where}\\
&\quad \mathit{modify} :: \mathit{rec}\ (\lambda \mathit{val} \to (\mathit{val} \to \mathit{val})) \to\\
&\qquad\qquad \mathit{rec}\ (\lambda \mathit{val} \to \mathit{val}) \to\\
&\qquad\qquad \mathit{rec}\ (\lambda \mathit{val} \to \mathit{val})\\
&\quad \textbf{instance } \mathit{Record}\ X \textbf{ where } \ldots\\
&\quad \textbf{instance } (\mathit{Record}\ \mathit{rec}) \Rightarrow\\
&\qquad\qquad \mathit{Record}\ (\mathit{rec} :\& \mathit{name} ::: \mathit{val}) \textbf{ where } \ldots
\end{aligned}$$

- problem with this approach:

  set of combinators is fixed

A fold for record schemes

- implementation of *modify* uses induction on record schemes
- capture the induction principle with a fold on record schemes:

```
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)
```

Implementation of *modify* using *fold*

- replacement type for the *thing* variable:

$$\begin{aligned}\textbf{type}\ \mathit{ModifyThing}\ \mathit{rec} = \mathit{rec}\ (\lambda \mathit{val} \to (\mathit{val} \to \mathit{val})) \to \\ \mathit{rec}\ (\lambda \mathit{val} \to \mathit{val}) \to \\ \mathit{rec}\ (\lambda \mathit{val} \to \mathit{val})\end{aligned}$$

- implementation of *modify*:

$$\begin{aligned}&\mathit{modify} = \mathit{fold}\ \mathit{nilModify}\ \mathit{snocModify}\ \textbf{where} \\ &\quad \mathit{nilModify} :: \mathit{ModifyThing}\ X \\ &\quad \mathit{nilModify}\ X\ X = \ldots \\ &\quad \mathit{snocModify} :: (\mathit{Record}\ \mathit{rec}) \Rightarrow \\ &\qquad \mathit{ModifyThing}\ \mathit{rec} \to \\ &\qquad \mathit{ModifyThing}\ (\mathit{rec} :\& \mathit{name} ::: \mathit{sort}) \\ &\quad \mathit{snocModify}\ \mathit{modify} \\ &\qquad (\mathit{modRec} :\& \mathit{name} := \mathit{mod}) \\ &\qquad (\mathit{rec} :\& \_ := \mathit{val}) = \ldots\end{aligned}$$

Demand for more generalization

- another example of a record combinator:

  converting records of arrays into records of lists

- presumed type:

$$(Record\ rec) \Rightarrow rec\ (\lambda array \longrightarrow array) \longrightarrow rec\ (\lambda(Array\ ix\ el) \longrightarrow [el])$$

- sorts have to be array types
- conflicts with the *Record* class:
  - *Record* class allows arbitrary types of kind $*$ as sorts
  - (:&)-alternative of *fold* must work with all sorts of kind $*$:

$$\forall rec\ name\ (sort :: *).(Record\ rec) \Rightarrow thing\ rec \longrightarrow thing\ (rec :\& name ::: sort)$$

Emulation of subkinds

- solution:

  allow arbitrary subkinds of $*$ as kind of sorts

- represent subkinds as types:

  data $Kind_{Array}$

- use a type class to specify inhabitants of kinds:

  class $Inhabitant\ kind\ \ sort$

  instance $(Ix\ ix) \Rightarrow Inhabitant\ Kind_{Array}\ (Array\ ix\ el)$

Kind-polymorphic *Record* class

- *Record* class parameterized by the kind of sorts:

```
class Record kind rec where
  fold :: thing X                                              →
          (∀rec name sort.
           (Record kind rec, Inhabitant kind sort) ⇒
           thing rec → thing (rec :& name ::: sort))           →
          thing rec
instance Record kind X                                  where ...
instance (Record kind rec,
          Inhabitant kind sort) ⇒
          Record kind (rec :& name ::: sort) where ...
```

Closing kinds

- problem:

  new instances of *Inhabitant* can be added anytime

- idea:

  enforce that for any type-level function F,

$$\forall sort.(Inhabitant\ Kind_{Array}\ sort) \Rightarrow F\ sort$$
$$\cong$$
$$\forall ix\ el.(Ix\ ix) \Rightarrow F\ (Array\ ix\ el)$$

- force the user to implement methods that convert forward and backwards between these types
- not only for $Kind_{Array}$ but analogously for any kind

Generic Record Combinators with Static Type Checking

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

12th International ACM SIGPLAN Symposium
on Principles and Practice of Declarative Programming

July 26–28, 2010

Is it really a fold?

- compare it to fold on lists
- heads of non-empty lists and complete list show up as function arguments:

$$thing \longrightarrow (el \longrightarrow thing \longrightarrow thing) \longrightarrow [el] \longrightarrow thing$$

- analogies between both folds:

$$\text{head} \longleftrightarrow \text{name and sort of last field}$$
$$\text{complete list} \longleftrightarrow \text{complete record scheme}$$

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

Yes, it is!

- applying equivalences to the type of *fold*:
  - inverse of forall hoisting:

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

  - from universal quantification to dependent types:

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

- transformation result:

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

Implementation of kind closing (1)

- for every kind, give the specific form of universal quantification and the forward conversion:

$$\textbf{class } \mathit{Kind}\ \mathit{kind} \textbf{ where}$$
$$\quad \textbf{data } \mathit{All}\ \mathit{kind} :: (* \to *) \to *$$
$$\quad \mathit{closed} :: (\forall \mathit{sort}.(\mathit{Inhabitant}\ \mathit{kind}\ \mathit{sort}) \Rightarrow \mathit{item}\ \mathit{sort}) \to \mathit{All}\ \mathit{kind}\ \mathit{item}$$

- specifically for $\mathit{Kind}_{\mathit{Array}}$:

$$\textbf{instance } \mathit{Kind}\ \mathit{Kind}_{\mathit{Array}} \textbf{ where}$$
$$\quad \textbf{data } \mathit{All}\ \mathit{Kind}_{\mathit{Array}}\ \mathit{item} = \mathit{All}_{\mathit{Array}}\ (\forall \mathit{ix}\ \mathit{el}.(\mathit{Ix}\ \mathit{ix}) \Rightarrow \mathit{item}\ (\mathit{Array}\ \mathit{ix}\ \mathit{el}))$$
$$\quad \mathit{closed}\ \mathit{item} = \mathit{All}_{\mathit{Array}}\ \mathit{item}$$

Implementation of kind closing (2)

- backwards conversion should have the type

$$All\ kind\ item \longrightarrow \forall sort.(Inhabitant\ kind\ sort) \Rightarrow item\ sort$$

- forall hoisting leads to

$$\forall sort.(Inhabitant\ kind\ sort) \Rightarrow All\ kind\ item \longrightarrow item\ sort$$

- make backwards conversion a method of *Inhabitant*:

$$\textbf{class}\ Inhabitant\ kind\ sort\ \textbf{where}$$
$$specialize :: All\ kind\ item \longrightarrow item\ sort$$

- implementation for $Kind_{Array}$:

$$\textbf{instance}\ (Ix\ ix) \Rightarrow$$
$$Inhabitant\ Kind_{Array}\ (Array\ ix\ el)\ \textbf{where}$$
$$specialize\ (All_{Array}\ item) = item$$