A Generic Foundation for Record Combinators

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

21st International Symposium
on Implementation and Application of Functional Languages

September 23–25, 2009

Simple DIY record system

- records as lists of name-value pairs:

$$\begin{aligned}\textbf{data}\ X &= X\\ \textbf{data}\ rec :\&\ field &= rec :\&\ field\\ \textbf{data}\ name ::: val &= name := val\end{aligned}$$

- field names represented by type constructor and data constructor:

$$\textbf{data}\ name = name$$

Example record

- field names:

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

- record:

$$\begin{aligned}
&\mathit{example} :: X \mathbin{:\&} \mathit{Surname} ::: \mathit{String}\\
&\qquad\qquad\quad \mathbin{:\&} \mathit{Age} ::: \mathit{Int}\\
&\qquad\qquad\quad \mathbin{:\&} \mathit{Room} ::: \mathit{String}\\
&\mathit{example} = X \mathbin{:\&} \mathit{Surname} := \text{"Jeltsch"}\\
&\qquad\qquad\quad \mathbin{:\&} \mathit{Age} := 31\\
&\qquad\qquad\quad \mathbin{:\&} \mathit{Room} := \text{"EH/202"}
\end{aligned}$$

Modification

- record of changes:

$$
\begin{aligned}
\mathit{changes} &:: X \mathrel{:\&} \mathit{Surname} ::: \mathit{String} \to \mathit{String} \\
& \mathrel{:\&} \mathit{Age} ::: \mathit{Int} \to \mathit{Int} \\
& \mathrel{:\&} \mathit{Room} ::: \mathit{String} \to \mathit{String} \\
\mathit{changes} &= X \mathrel{:\&} \mathit{Surname} := \mathit{id} \\
& \mathrel{:\&} \mathit{Age} := (+1) \\
& \mathrel{:\&} \mathit{Room} := \mathit{const}\ \text{"HG/2.14"}
\end{aligned}
$$

- want a function that performs the modification:

$$\mathit{modify}\ \mathit{changes}\ \mathit{example}$$

- should work with all types of records

Goal of this work

- problem:
  - *modify* works globally
  - only field-based access with traditional record systems
    - no implementation for *modify*
    - no type for *modify*
- in this talk: a record system that overcomes these deficiencies
- no ad-hoc solution:
  - generic foundation for building various record combinators
  - *modify* as a special case

Specifying record type relationships

- record type built from two ingredients:

  scheme of form

  $$X \mathbin{:\&} name_1 ::: sort_1 \mathbin{:\&} \ldots \mathbin{:\&} name_n ::: sort_n$$

  style a type-level function
- value types of the record type:

  $$style\ sort_1, \ldots, style\ sort_n$$

- related record types from same scheme with different styles
- declaration of record scheme types:

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

The type of *modify*

- value types of data record used as sorts
- two record styles:

$$\begin{aligned} \text{plain} \quad & \lambda val \to val \\ \text{modification} \quad & \lambda val \to (val \to val) \end{aligned}$$

- class *Record* of all record schemes
- type of *modify*:

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

- problem: no λ-expressions at the type level

Emulation of type-level λ-expressions

- type-level functions represented by phantom types
- type synonym family that describes function application:

  type family *App fun arg*

- representation of type-level function $\lambda\alpha \to \tau$:

  data *F*

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

- new declaration of field type:

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

- technique known as defunctionalization

The type of *modify* with emulation

- representations of plain and modification style:

  data *PlainStyle*

  data *ModStyle*

  type instance *App PlainStyle val* = *val*

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

- type of *modify*:

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

Implementation of *modify* as a class method

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{Record}\ \mathit{rec}\ \textbf{where} \\
\quad \mathit{modify} :: \mathit{rec}\ \mathit{ModStyle} \rightarrow \\
\qquad\qquad \mathit{rec}\ \mathit{PlainStyle} \rightarrow \\
\qquad\qquad \mathit{rec}\ \mathit{PlainStyle} \\
\textbf{instance}\ \mathit{Record}\ X \qquad\qquad\qquad\qquad \textbf{where}\ \ldots \\
\textbf{instance}\ (\mathit{Record}\ \mathit{rec}) \Rightarrow \\
\qquad \mathit{Record}\ (\mathit{rec}\ {:\&}\ \mathit{name} ::: \mathit{val})\ \textbf{where}\ \ldots
\end{array}
$$

- problem: closed set of combinators

A generic record combinator

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

$$
\begin{aligned}
&\textbf{class}\ \mathit{Record}\ \mathit{rec}\ \textbf{where}\\
&\quad \mathit{fold} :: \mathit{thing}\ X \rightarrow\\
&\qquad\qquad (\forall \mathit{rec}\ \mathit{name}\ \mathit{sort}.(\mathit{Record}\ \mathit{rec}) \Rightarrow\\
&\qquad\qquad\ \ \mathit{thing}\ \mathit{rec} \rightarrow \mathit{thing}\ (\mathit{rec}\ {:}\&\ \mathit{name}\ {:::}\ \mathit{sort})) \rightarrow\\
&\qquad\qquad \mathit{thing}\ \mathit{rec}\\
&\textbf{instance}\ \mathit{Record}\ X\ \textbf{where}\\
&\quad \mathit{fold}\ \mathit{nilAlt}\ \_ = \mathit{nilAlt}\\
&\textbf{instance}\ (\mathit{Record}\ \mathit{rec}) \Rightarrow\\
&\qquad\qquad \mathit{Record}\ (\mathit{rec}\ {:}\&\ \mathit{name}\ {:::}\ \mathit{sort})\ \textbf{where}\\
&\quad \mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt} = \mathit{snocAlt}\ \$\\
&\qquad\qquad\qquad\qquad \mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt}
\end{aligned}
$$

Implementation of *modify* using *fold*

- *thing* corresponds to type of *modify*:

$$\mathbf{newtype}\ \mathit{ModThing}\ \mathit{rec} = \mathit{ModThing}\ (\mathit{rec}\ \mathit{ModStyle} \rightarrow \mathit{rec}\ \mathit{PlainStyle} \rightarrow \mathit{rec}\ \mathit{PlainStyle})$$

- actual implementation:

$$\mathit{modify} = \mathbf{case}\ \mathit{fold}\ \mathit{nilAlt}\ \mathit{snocAlt}\ \mathbf{of}\ \mathit{ModThing}\ \mathit{comb} \rightarrow \mathit{comb}\ \mathbf{where}$$

$$\mathit{nilAlt} :: \mathit{ModThing}\ X$$

$$\mathit{nilAlt} = \ldots$$

$$\mathit{snocAlt} :: (\mathit{Record}\ \mathit{rec}) \Rightarrow \mathit{ModThing}\ \mathit{rec} \rightarrow \mathit{ModThing}\ (\mathit{rec}\ {:}\&\ \mathit{name}\ {:::}\ \mathit{val})$$

$$\mathit{snocAlt} = \ldots$$

Signal records in the Grapefruit FRP library

- signals have types of the form *sig era val*
- want records of signals with common era
- era goes into the style (unique for the whole record):

$$\textbf{data}\ \mathit{SignalStyle}\ \mathit{era}$$

- sorts are pairs of a signal type and a value type:

$$\textbf{data}\ (\mathit{sig} :: * \to * \to *)\ \text{`}\mathit{Of}\text{`}\ (\mathit{val} :: *)$$

- style application adds the era:

$$\begin{aligned}\textbf{type instance}\ \mathit{App}\ & (\mathit{SignalStyle}\ \mathit{era}) \\ & (\mathit{sig}\ \text{`}\mathit{Of}\text{`}\ \mathit{val}) \quad = \mathit{sig}\ \mathit{era}\ \mathit{val}\end{aligned}$$

- example of a signal record scheme:

$$\begin{aligned}(X & :\&\ \mathit{Position} ::: \mathit{CSignal}\ \text{`}\mathit{Of}\text{`}\ \mathit{Point} \\ & :\&\ \mathit{IsActive} ::: \mathit{SSignal}\ \text{`}\mathit{Of}\text{`}\ \mathit{Bool})\end{aligned}$$

Subkinds for sorts

- *Record* class allows arbitrary types of kind $*$ as sorts
- (:&)-alternative of *fold* must work with all sorts of kind $*$:

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

- problem: signal-related record combinators only work with sorts of the form *sig* ‘*Of*‘ *val*
- idea: allow arbitrary subkinds of $*$ as kind of sorts
- represent such subkinds as types:

data *SigOfVal*

- use a type class to specify inhabitants of kinds:

class *Inhabitant kind sort*
instance *Inhabitant SigOfVal* (*sig* ‘*Of*‘ *val*)

Kind-polymorphic *Record* class

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{Record}\ \mathit{kind}\ \mathit{rec}\ \textbf{where} \\
\quad \mathit{fold} :: \mathit{thing}\ X \rightarrow \\
\qquad (\forall \mathit{rec}\ \mathit{name}\ \mathit{sort}. \\
\qquad\ (\mathit{Record}\ \mathit{kind}\ \mathit{rec}, \mathit{Inhabitant}\ \mathit{kind}\ \mathit{sort}) \Rightarrow \\
\qquad\ \mathit{thing}\ \mathit{rec} \rightarrow \mathit{thing}\ (\mathit{rec} :\&\ \mathit{name} ::: \mathit{sort})) \rightarrow \\
\qquad \mathit{thing}\ \mathit{rec} \\
\textbf{instance}\ \mathit{Record}\ \mathit{kind}\ X\ \textbf{where}\ \ldots \\
\textbf{instance}\ (\mathit{Record}\ \mathit{kind}\ \mathit{rec}, \\
\qquad \mathit{Inhabitant}\ \mathit{kind}\ \mathit{sort}) \Rightarrow \\
\qquad \mathit{Record}\ \mathit{kind}\ (\mathit{rec} :\&\ \mathit{name} ::: \mathit{sort})\ \textbf{where}\ \ldots
\end{array}
$$

Closing kinds

- have to give the (:&)-alternative for all *sort* with *Inhabitant SigOfVal sort*
- problem: new instances of *Inhabitant* can be added anytime
- idea: enforce that for any $\mathit{item} :: * \longrightarrow *$,

$$\forall \mathit{sort}.(\mathit{Inhabitant}\ \mathit{SigOfVal}\ \mathit{sort}) \Rightarrow \mathit{item}\ \mathit{sort}$$

$$\cong$$

$$\forall (\mathit{sig} :: * \longrightarrow * \longrightarrow *)\ (\mathit{val} :: *).\mathit{item}\ (\mathit{sig}\ `\mathit{Of}`\ \mathit{val})$$

- force the user to implement methods that convert forward and backwards between these types
- not only for *SigOfVal* but analogously for any kind

Implementation of kind closing (1)

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

  class *Kind kind* **where**

  data *Forall kind* $:: (* \to *) \to *$

  $encase :: (\forall sort.(Inhabitant\ kind\ sort) \Rightarrow item\ sort) \to Forall\ kind\ item$

- specifically for *SigOfVal*:

  type $ForallSOV\ item = \forall (sig :: * \to * \to *)\ (val :: *).\ item\ (sig\ `Of`\ val)$

  instance *Kind SigOfVal* **where**

  data $Forall\ SigOfVal\ item = Forall\ (ForallSOV\ item)$

  $encase\ item = Forall\ item$

Implementation of kind closing (2)

- backwards conversion should have the type

$$\mathit{Forall}\ \mathit{kind}\ \mathit{item} \to \forall \mathit{sort}.(\mathit{Inhabitant}\ \mathit{kind}\ \mathit{sort}) \Rightarrow \mathit{item}\ \mathit{sort}$$

- forall hoisting leads to

$$\forall \mathit{sort}.(\mathit{Inhabitant}\ \mathit{kind}\ \mathit{sort}) \Rightarrow \mathit{Forall}\ \mathit{kind}\ \mathit{item} \to \mathit{item}\ \mathit{sort}$$

- make backwards conversion a method of *Inhabitant*:

class *Inhabitant kind sort* **where**
  specialize :: *Forall kind item* → *item sort*

- implementation for *SigOfVal*:

instance *Inhabitant SigOfVal* (*sig* ‘*Of*‘ *val*) **where**
  specialize (*Forall item*) = *item*

A Generic Foundation for Record Combinators

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

21st International Symposium
on Implementation and Application of Functional Languages

September 23–25, 2009

Is it really a fold?

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

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

- 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*:
  - from universal quantification to dependent types:

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

  - inverse of forall hoisting:

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

- transformation result:

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

Signals in the Grapefruit FRP library

- signals describe temporal behavior
- different types of signals:

discrete	*DSignal*
segmented	*SSignal*
continuous	*CSignal*

- all signal types have two parameters (of kind $*$):

era	phantom parameter that denotes the lifetime of the signal
val	value space of the signal