First-Class Subkinds in Haskell

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 27, 2011

Overview

Recapitulation

Subkinds needed

Subkind emulation

Closing subkinds

Overview

Recapitulation

Subkinds needed

Subkind emulation

Closing subkinds

Record types

- record type is an application of a record scheme to a style parameter, where the style denotes a type-level function:

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

$$\textbf{data}\ (\rho :\& \varphi)\ \sigma = \rho\ \sigma :\& \varphi\ \sigma$$

- type synonym family *App* describes type-level function application:

$$\textbf{type family}\ \mathit{App}\ \varphi\ \alpha$$

- field scheme consists of a name and a sort:

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

- families of related record types can be generated by applying the same scheme to different styles

Record scheme induction

class *Record* ρ **where**

$$fold :: \theta\ X \rightarrow (\forall \rho\ \nu\ \varsigma\ .\ (Record\ \rho) \Rightarrow \theta\ \rho \rightarrow \theta\ (\rho :\&\ \nu ::: \varsigma)) \rightarrow \theta\ \rho$$

instance *Record X* **where**

$$fold\ f_X\ \_ = f_X$$

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

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

The *modify* combinator

- combinator for modifying all fields of a record:

$$modify\ (X :\&\ \nu_1 := f_1\ :\&\ \ldots :\&\ \nu_n := f_n)\ (X :\&\ \nu_1 := x_1 :\&\ \ldots :\&\ \nu_n := x_n)$$

$$=$$

$$X :\&\ \nu_1 := f_1\ x_1 :\&\ \ldots :\&\ \nu_n := f_n\ x_n$$

- type of *modify*:

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

- record styles used in the type of *modify*:

$$\textbf{type instance}\ App\ \Sigma_{Plain}\ \alpha = \alpha$$

$$\textbf{type instance}\ App\ \Sigma_{Mod}\ \alpha = \alpha \rightarrow \alpha$$

- *modify* implemented as a *fold* application, using the following replacement for the θ-variable:

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

Overview

Recapitulation

Subkinds needed

Subkind emulation

Closing subkinds

The *mapElems* combinator

- conversion from arrays to lists:

$$elems :: (Ix\ \iota) \Rightarrow Array\ \iota\ \alpha \to [\alpha]$$

- conversion from array records to list records:

$$mapElems\ (X :\&\ \nu_1 := a_1 :\& \ldots :\&\ \nu_n := a_n)$$
$$=$$
$$X :\&\ \nu_1 := elems\ a_1 :\& \ldots :\&\ \nu_n := elems\ a_n$$

- sorts must denote pairs of an index and an element type
- pick some type constructor Π and encode (ι, α) as $\Pi\ \iota\ \alpha$
- let's choose *Array* as our Π
- styles for the record argument and the list result:

$$\textbf{type instance}\ App\ \Sigma_{Array}\ (Array\ \iota\ \alpha) = Array\ \iota\ \alpha$$
$$\textbf{type instance}\ App\ \Sigma_{List}\ (Array\ \iota\ \alpha) = [\alpha]$$

- type of *mapElems*:

$$(Record\ \rho) \Rightarrow \rho\ \Sigma_{Array} \to \rho\ \Sigma_{List}$$

Problems with this combinator

- *mapElems* can only work with sorts that are array types
- so the type

  $$(Record\ \rho) \Rightarrow \rho\ \Sigma_{Array} \rightarrow \rho\ \Sigma_{List}$$

  is too general
- implementation of *mapElems* based on *fold*:

  $$mapElems = fold\ f_X\ f_{(:\&)}\ \textbf{where}$$
  $$f_X \quad = \lambda X \qquad \rightarrow X$$
  $$f_{(:\&)}\ g = \lambda (r :\&\ \nu := a) \rightarrow g\ r :\&\ \nu := elems\ a$$
- problem with this implementation:
  - most general type of $f_{(:\&)}$:

    $$\forall \rho\ \nu\ \iota\ \alpha\ .\ (Record\ \rho, Ix\ \iota) \Rightarrow$$
    $$\Theta_{mapElems}\ \rho \rightarrow \Theta_{mapElems}\ (\rho :\&\ \nu ::: Array\ \iota\ \alpha)$$
  - required type:

    $$\forall \rho\ \nu\ \varsigma\ .\quad (Record\ \rho) \Rightarrow$$
    $$\Theta_{mapElems}\ \rho \rightarrow \Theta_{mapElems}\ (\rho :\&\ \nu ::: \varsigma)$$

Subkinds to the rescue

- introduce subkinds as the kind-level analog of subtypes
- subtypes of kind $*$ cover some types of kind $*$
- examples of such subkinds:

Array	all types $\mathit{Array}\ \iota\ \alpha$ with $(\mathit{Ix}\ \iota)$
Map	all types $\mathit{Map}\ \kappa\ \alpha$ with $(\mathit{Ord}\ \kappa)$ and all types $\mathit{IntMap}\ \alpha$
$*$	all types of kind $*$

- extend the *Record* class such that it allows for induction over all schemes whose sorts are of a given subkind
- define *mapElems* such that it only works for sorts of kind *Array*
- problem:

  no support for subkinds in present-day Haskell

Overview

Emulation of subkinds

- represent subkinds by types:

  $$\textbf{data}\ \Xi_{\mathit{Array}}$$
  $$\textbf{data}\ \Xi_{\mathit{Map}}$$
  $$\textbf{data}\ \Xi_{*}$$

- ordinary parametric polymorphism can be used to emulate subkind polymorphism
- class *Inhabitant* that specifies subkind inhabitation:

$$\begin{array}{llll}
\textbf{class} & \mathit{Inhabitant}\ \xi & \varsigma \\
\textbf{instance}\ (\mathit{Ix}\ \iota) \Rightarrow & \mathit{Inhabitant}\ \Xi_{\mathit{Array}} & (\mathit{Array}\ \iota\ \alpha) \\
\textbf{instance}\ (\mathit{Ord}\ \kappa) \Rightarrow & \mathit{Inhabitant}\ \Xi_{\mathit{Map}} & (\mathit{Map}\ \kappa\ \alpha) \\
\textbf{instance} & \mathit{Inhabitant}\ \Xi_{\mathit{Map}} & (\mathit{IntMap}\ \alpha) \\
\textbf{instance} & \mathit{Inhabitant}\ \Xi_{*} & \alpha
\end{array}$$

Subkind support for records

- *Record* class with subkind support:

$$
\begin{aligned}
&\textbf{class } \mathit{Record}\ \xi\ \rho\ \textbf{where} \\
&\quad \mathit{fold} :: \theta\ X \rightarrow \\
&\qquad (\forall \rho\ \nu\ \varsigma\ .\ (\mathit{Record}\ \xi\ \rho, \mathit{Inhabitant}\ \xi\ \varsigma) \Rightarrow \\
&\qquad\qquad \theta\ \rho \rightarrow \theta\ (\rho :\&\ \nu ::: \varsigma)) \rightarrow \\
&\qquad \theta\ \rho \\
&\textbf{instance } \mathit{Record}\ \xi\ X\ \textbf{where} \\
&\quad \mathit{fold}\ f_X\ \_ \quad = f_X \\
&\textbf{instance } (\mathit{Record}\ \xi\ \rho, \mathit{Inhabitant}\ \xi\ \varsigma) \Rightarrow \\
&\qquad \mathit{Record}\ \xi\ (\rho :\&\ \nu ::: \varsigma)\ \textbf{where} \\
&\quad \mathit{fold}\ f_X\ f_{(:\&)} = f_{(:\&)}\ (\mathit{fold}\ f_X\ f_{(:\&)})
\end{aligned}
$$

- type of *mapElems*:

$$(\mathit{Record}\ \Xi_{\mathit{Array}}\ \rho) \Rightarrow \rho\ \Sigma_{\mathit{Array}} \rightarrow \rho\ \Sigma_{\mathit{List}}$$

A problem with open classes

- from the definition of *mapElems*:

$$f_{(:\&)}\ g = \lambda(r :\&\ \nu := a) \rightarrow g\ r :\&\ \nu := \mathit{elems}\ a$$

- most general type of $f_{(:\&)}$:

$$\forall \rho\ \nu\ \iota\ \alpha\ .\ (\mathit{Record}\ \rho, \mathit{Ix}\ \iota) \Rightarrow$$
$$\Theta_{\mathit{mapElems}}\ \rho \rightarrow \Theta_{\mathit{mapElems}}\ (\rho :\&\ \nu ::: \mathit{Array}\ \iota\ \alpha)$$

- required type:

$$\forall \rho\ \nu\ \varsigma\ .\quad (\mathit{Record}\ \rho, \mathit{Inhabitant}\ \Xi_{\mathit{Array}}\ \varsigma) \Rightarrow$$
$$\Theta_{\mathit{mapElems}}\ \rho \rightarrow \Theta_{\mathit{mapElems}}\ (\rho :\&\ \nu ::: \varsigma)$$

- required type still more general, since *Inhabitant* could be extended:

instance $\mathit{Inhabitant}\ \Xi_{\mathit{Array}}\ \mathit{Bool}$

Overview

Recapitulation

Subkinds needed

Subkind emulation

Closing subkinds

Closing the *Array* subkind

- enforce that the set of all ς with $(\mathit{Inhabitant}\ \Xi_{\mathit{Array}}\ \varsigma)$ is the set of all *Array* inhabitants
- realized by enforcing that

$$\forall \varsigma\ .\ (\mathit{Inhabitant}\ \Xi_{\mathit{Array}}\ \varsigma) \Rightarrow F\ \varsigma$$
$$\cong$$
$$\forall \varsigma :: \mathit{Array}\ .\ F\ \varsigma$$

  for all type-level functions F
- sufficient to enforce this for all types $F :: * \to *$, since for every type-level function there is an isomorphic newtype wrapper
- expressing universal quantification over *Array* inhabitants in Haskell:

$$\forall \varsigma :: \mathit{Array}\ .\ F\ \varsigma$$
$$=$$
$$\forall \iota\ \alpha\ .\ (\mathit{Ix}\ \iota) \Rightarrow F\ (\mathit{Array}\ \iota\ \alpha)$$

Universal quantification over *Map* inhabitants

- *Map* is the sum of two subkinds:

Map_1	all types $Map\ \kappa\ \alpha$ with $(Ord\ \kappa)$
Map_2	all types $IntMap\ \alpha$

- universal quantification can be expressed for both subkinds:

$$(\forall\varsigma :: Map_1\ .\ F\ \varsigma) = (\forall\kappa\ \alpha\ .\ (Ord\ \kappa) \Rightarrow F\ (Map\ \kappa\ \alpha))$$
$$(\forall\varsigma :: Map_2\ .\ F\ \varsigma) = (\forall\alpha\ .\ F\ (IntMap\ \alpha))$$

- expressing universal quantification for *Map*:

$$\forall\varsigma :: Map\ .\ F\ \varsigma$$
$$=$$
$$(\forall\kappa\ \alpha\ .\ (Ord\ \kappa) \Rightarrow F\ (Map\ \kappa\ \alpha), \forall\alpha\ .\ F\ (IntMap\ \alpha))$$

Closing arbitrary subkinds

- introduce a data family *All* that denotes universal quantification for all subkind representations:

  $$\textbf{data family } \mathit{All}\ \xi :: (* \to *) \to *$$

- add an instance declaration for every subkind:

  $$\begin{aligned}&\textbf{data instance}\\ &\quad \mathit{All}\ \Xi_{\mathit{Array}}\ \varphi = \mathit{All}_{\mathit{Array}}\ (\forall \iota\ \alpha\ .\ (\mathit{Ix}\ \iota) \Rightarrow \varphi\ (\mathit{Array}\ \iota\ \alpha))\\ &\textbf{data instance}\\ &\quad \mathit{All}\ \Xi_{\mathit{Map}}\ \varphi = \mathit{All}_{\mathit{Map}}\ (\forall \kappa\ \alpha\ .\ (\mathit{Ord}\ \kappa) \Rightarrow \varphi\ (\mathit{Map}\ \kappa\ \alpha))\ (\forall \alpha\ .\ \varphi\ (\mathit{IntMap}\ \alpha))\\ &\textbf{data instance}\\ &\quad \mathit{All}\ \Xi_{*}\ \varphi = \mathit{All}_{*}\ (\forall \alpha\ .\ \varphi\ \alpha)\end{aligned}$$

- enforce the isomorphism

  $$(\forall \varsigma\ .\ (\mathit{Inhabitant}\ \xi\ \varsigma) \Rightarrow \varphi\ \varsigma) \cong (\mathit{All}\ \xi\ \varphi)$$

  by requiring the implementation of conversion functions

Forward conversion

- introduce a class of all subkind representations with a method for forward conversion:

  $$\textbf{class } \mathit{Kind}\ \xi\ \textbf{where}$$
  $$\mathit{closed} :: (\forall\varsigma\ .\ (\mathit{Inhabitant}\ \xi\ \varsigma) \Rightarrow \varphi\ \varsigma) \to \mathit{All}\ \xi\ \varphi$$

- instance declarations:

  $$\textbf{instance } \mathit{Kind}\ \Xi_{\mathit{Array}}\ \textbf{where}$$
  $$\mathit{closed}\ x = \mathit{All}_{\mathit{Array}}\ x$$
  $$\textbf{instance } \mathit{Kind}\ \Xi_{\mathit{Map}}\ \textbf{where}$$
  $$\mathit{closed}\ x = \mathit{All}_{\mathit{Map}}\ x\ x$$
  $$\textbf{instance } \mathit{Kind}\ \Xi_{*}\ \textbf{where}$$
  $$\mathit{closed}\ x = \mathit{All}_{*}\ x$$

- add a context to the *Record* class:

  $$\textbf{class } (\mathit{Kind}\ \xi) \Rightarrow \mathit{Record}\ \xi\ \rho\ \textbf{where}\ \ldots$$

Backwards conversion

- add a context and a method for backwards conversion to the *Inhabitant* class:

  class $(\mathit{Kind}\ \xi) \Rightarrow \mathit{Inhabitant}\ \xi\ \varsigma$ **where**

  $\quad \mathit{specialize} :: \mathit{All}\ \xi\ \varphi \to \varphi\ \varsigma$

- instance declarations:

  instance $(\mathit{Ix}\ \iota) \Rightarrow$
  $\qquad \mathit{Inhabitant}\ \Xi_{\mathit{Array}}\ (\mathit{Array}\ \iota\ \alpha)$ **where**
  $\quad \mathit{specialize}\ (\mathit{All}_{\mathit{Array}}\ x) = x$

  instance $(\mathit{Ord}\ \kappa) \Rightarrow$
  $\qquad \mathit{Inhabitant}\ \Xi_{\mathit{Map}}\ (\mathit{Map}\ \kappa\ \alpha)$ **where**
  $\quad \mathit{specialize}\ (\mathit{All}_{\mathit{Map}}\ x\ \_) = x$

  instance $\mathit{Inhabitant}\ \Xi_{\mathit{Map}}\ (\mathit{IntMap}\ \alpha)$ **where**
  $\quad \mathit{specialize}\ (\mathit{All}_{\mathit{Map}}\ \_\ x) = x$

  instance $\mathit{Inhabitant}\ \Xi_{*}\ \alpha$ **where**
  $\quad \mathit{specialize}\ (\mathit{All}_{*}\ x) = x$

First-Class Subkinds in Haskell

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 27, 2011