Generic Record Combinators with Static Type Checking

Wolfgang Jeltsch [†]

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau
Postfach 10 13 44, 03013 Cottbus, Germany
jeltsch@tu-cottbus.de

Abstract

Common record systems only provide access to individual record fields. However, it is often useful to have generic record combinators, that is, functions that work with complete records of varying structure. Traditionally, generic record combinators can only be implemented in dynamically typed languages.

In this paper, we present a record system for Haskell that allows us to define generic record combinators without giving up static type checks. The system is implemented as a library so that it works without modifications to the language. We achieve this mainly by using advanced type system features such as type equality constraints, type families, and higher-rank polymorphism. A key contribution of our paper is a technique for emulating subkinds, including subkind polymorphism. We use this to give the record system additional expressiveness.

Our system is not only useful in itself. It also shows what features should be taken into account when designing language support for records.

1. Introduction

Traditional record systems of statically typed languages focus on working with single record fields. As a minimum, they provide operations for field selection and modification. Systems with support for extensible records also permit the user to add and remove fields. A typical example of such a system is the one Jones and Peyton Jones proposed for Haskell [7].

In that system, a record is a mapping from field names to values, and the type of a record is a mapping from the record's field names to the types of the corresponding values. For example, the expression

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

denotes a record that maps the names $surname$, age, and $room$ to the values $\texttt{"Jeltsch"}$, 31, and $\texttt{"EH/202"}$, respectively. The type of this record is

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

Say we want to modify the above record. The age shall be incremented by two, and the room has to be replaced by $\texttt{"HG/2.39"}$. It would be nice to specify these modifications also as a record, namely,

$$\{\, age = (+2), room = const \ \texttt{"HG/2.39"} \,\} \ ,$$

which has the type

$$\{\, age :: Int \rightarrow Int, room :: String \rightarrow String \,\} \ .$$

The field values of this modification record are functions that shall be applied to field values of the above data record. The modification record uses field names from the data record to associate the functions with their corresponding values.

To actually perform the update, we want to have a function $modify$. Applying $modify$ to a modification record and a data record shall yield the modified data record. The $modify$ function shall work with data records of arbitrary type. It shall accept all modification records whose type fits the type of the data record. There are two things to note:

- The modification function has to iterate through all fields of the modification record. It is not enough to access a fixed number of fields using statically known names.

- The modification record must only use field names that occur in the data record, and a value assigned to such a name must have type $\tau \rightarrow \tau$ if the corresponding value from the data record has type τ. This should be statically checked and therefore expressed by the type of the modification function.

Because of these conditions, it is impossible to implement $modify$ in record systems like the one of Jones and Peyton Jones. Of course, $modify$ is implementable in dynamically typed languages. In such languages, we can represent records by, for example, ordinary lists whose elements are name-value pairs. We can then use list induction to implement $modify$. However, we do not want to give up static type checking. Therefore, we develop a record system that allows us to implement $modify$ and similar functions without resorting to runtime checks. We implement our system as a Haskell library, using only language extensions that are already supported by the Glasgow Haskell Compiler (GHC).

[†] I dedicate this work to my father Hans-Joachim Jeltsch who passed away when I was writing this paper.

We start with a simple implementation of heterogenous lists of name-value pairs in section 2. Building on this foundation, we make the following contributions:

- We introduce the concept of record type families in section 3, along with an implementation. A record type family is an indexed set of related record types that is generated from a so-called record scheme. We specify a restricted variant of the modification function and show that record type families make it possible to give a type to this function.

- In section 4, we define a fold operator that allows us to define record combinators via induction over record schemes. We present an implementation of the restricted modification function based on this fold operator.

- In section 5, we implement record conversion, which allows us to abstract from irrelevant order of fields, and to ignore fields we are not interested in. Our implementation uses type equality constraints to control class instance selection in the presence of overlapping instances. We show that we can implement the modification function in full generality using record conversion. We also demonstrate that record conversion can be used to support pattern matching for records.

- In section 6, we present a technique for emulating subkinds, including subkind polymorphism. We use this emulation to generalize the above-mentioned fold operator. As a result, a larger class of record combinators can be implemented.

We compare our record system to related work in section 7. Finally, we give conclusions and an outlook on further work in section 8.

The complete record system is implemented in the Haskell packages *records* [4], *kinds* [3], and *type-functions* [6]. Our developments were driven by actual practical demands that showed up while working on the Grapefruit library [5].

2. A Simple Record Library

We first show a simple implementation of records as lists of name-value pairs. The types of these lists specify the names of the record fields along with the types of the corresponding values. We develop our implementation bottom-up, starting with the representation of field names.

We have to represent names both on the type level and on the value level. For each name, we declare a nullary type constructor with a single nullary data constructor that uses the same identifier as the type constructor. So we declare the names used in the introductory example as follows:

data $Surname = Surname$
data $Age \quad = Age$
data $Room \quad = Room$

A record field is a pair of a name and a value. So we could use ordinary pairs to represent fields. We do not do that since it would lead to shabby syntax.[1] Instead, we declare a special field type:

data $name ::: val = name := val$

The operator symbol ::: was chosen because it is similar to the special symbol ::, which stands for "has type".

We define records as heterogenous lists of fields. We introduce two constructors, one for the empty record and one for record extension:

---
[1] The actual implementation in the *records* package also makes field names strict, which is another reason for not using the ordinary pair type.

data $X \qquad = X$
data $rec :\& field = rec :\& field$

While we could use the unit type () and the pair type (,) instead of X and (:&), we chose to introduce new types for similar reasons as we did for record fields. Note that (:&) is a "snoc", not a "cons", that is, new fields are appended, not prepended. In the following, we assume that :& is left-associative and of higher priority than ::: and :=.

The example record from the introduction can now be written

$X:\&Surname:=\texttt{"Jeltsch"}:\&Age:=31:\&Room:=\texttt{"EH/202"}$.

The type of this record is

$X :\& Surname ::: String :\& Age ::: Int :\& Room ::: String$.

Records according to the above definition are not name-to-value maps. There are the following issues:

- A name may occur multiple times in the same record. Since this can even be an advantage [9], we retain this property.

- The fields of a record are ordered. Having an order is good for fields with the same name since it allows us to distinguish such fields by relative position. However, it introduces redundancy otherwise. We show how we can overcome the redundancy problem in section 5.

3. Record Type Families

We now discuss record type families, which allow us to specify certain relationships between record types. Record type families alone are not enough for giving the modification function a proper type. However, we can find a type for a restricted version of $modify$. The restriction is that the modification record must contain a field for each field of the data record, and that those fields must occur in the same order as their data record counterparts. To modify the example record as described in the introduction, we have to use the modification record

$X:\&Surname:=id:\&Age:=(+2):\&Room:=const \texttt{"HG/2.39"}$

now. In this section and the next, we will only deal with the restricted $modify$.

3.1 Record Type Family Essentials

The restricted $modify$ function enforces a very strong relationship between the type of a modification record and a corresponding data record. The type of the data record uniquely determines the type of the modification record. We can generate the latter from the former by replacing every value type τ with $\tau \rightarrow \tau$. Record type families allow us to perform such transformations on the fly.

A record type family is characterized by a record scheme. A record scheme is a list of pairs, each consisting of a name and a so-called sort. A sort is a type, so record schemes have the same structure as our record types from the previous section. However, sorts are not used as value types directly. We build a record type by combining a record scheme with a type-level function, which is called the style of the record type. The record style is applied to all sorts of the scheme to generate the value types of the respective fields. By coupling the same scheme with different styles, we get a family of related record types.

We want to modify the declarations of X, (:&), and (:::) such that types of the form

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

denote record schemes with names ν_1 through ν_n and sorts ς_1 through ς_n. Applying such a type to a style shall yield the respective record type. The new declarations are as follows:

$$\textbf{data } X \qquad\qquad style = X$$
$$\textbf{data } (rec :\& field) \quad style = rec\ style :\& field\ style$$
$$\textbf{data } (name ::: sort)\ style = name := style\ sort$$

We define a class *Record* of all record schemes:

$$\textbf{class} \qquad\qquad\qquad Record\ rec$$
$$\textbf{instance} \qquad\qquad\quad Record\ X$$
$$\textbf{instance } (Record\ rec) \Rightarrow Record\ (rec :\& name ::: sort)$$

Now, we want to use record type families to give a type to *modify*. We generate the type of the modification record and the type of the data record from the same scheme. This scheme uses the value types of the data record as its sorts. So the style of the data record has to be the identity function, and the style of the modification record has to be the function that maps each type τ to $\tau \to \tau$. For now, let us assume that we had type-level abstractions available such that $\lambda\alpha \to \tau'$ denotes a type-level function that maps each type τ of kind $*$ to $\tau'[\tau/\alpha]$. The modification function has the type

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

This type allows to modify records of style $\lambda val \to val$ only. However, we can give a more general type to the *modify* function such that it accepts data records of arbitrary style. The new type is

$$(Record\ rec) \Rightarrow rec\ (\lambda sort \to (style\ sort \to style\ sort)) \to$$
$$rec\ style \qquad\qquad\qquad \to$$
$$rec\ style\ .$$

3.2 Emulation of Type-Level Abstractions

Unfortunately, Haskell does not support type-level abstractions. However, we can emulate them by using defunctionalization [10] at the type level. We represent type-level functions by ordinary types and introduce a type synonym family [11] that describes type-level function application:

$$\textbf{type family } App\ fun\ arg$$

Instances of *App* have to be defined such that for each type-level function F with representation φ and each argument type τ, the type $App\ \varphi\ \tau$ equals $F\ \tau$.

For each type-level abstraction $\lambda\alpha \to \tau'$ with free variables β_1 through β_n, we introduce an n-ary type constructor Λ whose arguments have the same kinds as β_1 through β_n. In addition, we add the following instance declaration for *App*:

$$\textbf{type instance } App\ (\Lambda\ \beta_1\ \dots\ \beta_n)\ \alpha = \tau'$$

Then, the type $\Lambda\ \beta_1\ \dots\ \beta_n$ represents the type-level function $\lambda\alpha \to \tau'$.

We have to modify the record-related types such that they use representations of type-level functions instead of the functions themselves. We can keep the data declarations of X and $(:\&)$ but have to change the data declaration of $(:::)$ to the following:

$$\textbf{data } (name ::: sort)\ style = name := App\ style\ sort$$

We will now formulate the type of *modify* using emulation of type-level abstractions. We introduce a type *ModStyle* for representing the styles of modification records:

$$\textbf{data } ModStyle\ style$$

The *App* instance declaration for *ModStyle* mirrors the abstraction $\lambda sort \to (App\ style\ sort \to App\ style\ sort)$:

$$\textbf{type instance}$$
$$App\ (ModStyle\ style)\ sort = App\ style\ sort \to$$
$$App\ style\ sort$$

Now, the type of the modification function is

$$(Record\ rec) \Rightarrow rec\ (ModStyle\ style) \to$$
$$rec\ style \qquad\qquad \to$$
$$rec\ style\ .$$

4. Record Scheme Induction

We can implement *modify* using induction on its record scheme parameter. Since record schemes are types, not values, we do not use pattern matching to distinguish between empty and non-empty record schemes. Instead, we make *modify* a method of the *Record* class and provide a method declaration for each of the two cases as part of the respective instance declaration. Figure 1 shows the complete code.

Of course, we also want to use record scheme induction to implement other combinators than *modify*. For each of them, we have three different options of implementing it:

1. We can add a new class that has the same instances as *Record* and contains the combinator as its method.

2. We can implement the combinator as a new method of the *Record* class or of one of the classes created according to option 1.

3. We can implement the combinator as an ordinary function.

If we use option 1, we end up with multiple classes that have the same instances, but the type checker cannot see that their instances are the same. So if we use multiple record combinators in a single expression, the type of the expression may contain lots of different class assertions that all mean the same thing. Therefore, we drop option 1.

Now, all inductively defined record combinators must be either methods of *Record* or ordinary functions. Once a class is declared, its set of methods is fixed. So we have to decide once and for all which combinators shall be implemented as methods of *Record* at the time we declare *Record*. Only these combinators can use record scheme induction directly by having different declarations for the two instance declarations of *Record*. All other combinators can use induction only indirectly by applying the methods of *Record*.

We declare a single method of *Record* that captures induction over record schemes in full generality. That way, every inductively defined record combinator can be implemented as an ordinary function that uses that method. Induction principles are represented by fold operators. So what we want is a fold over record schemes. We remove the *modify* method from the *Record* class and add a method *fold*. Figure 2 shows the resulting definition of *Record*.

We can produce an inductively defined record combinator by applying *fold* to an X-alternative and a $(:\&)$-alternative. These alternatives describe how specializations of the combinator for specific record schemes are constructed. The X-alternative is the specialization for the empty record scheme. The $(:\&)$-alternative produces specializations for non-empty record schemes $\rho :\& \nu ::: \varsigma$ from the specializations for the respective schemes ρ.

It is clear from figure 2 that the resulting combinator has the type $(Record\ rec) \Rightarrow \theta\ rec$ where θ is the type that is substituted for the variable *thing*. Since we cannot substitute arbitrary type-level functions for *thing*, most record combinators cannot be implemented as *fold* applications. For example, to implement *modify* as a result of *fold*, we would have to replace *thing* with

$$\lambda rec \to rec\ (ModStyle\ style) \to rec\ style \to rec\ style\ .$$

However, this type-level function is not a Haskell type.

```
class Record rec where
    modify :: rec (ModStyle style) → rec style → rec style
instance Record X where
    modify X X = X
instance (Record rec) ⇒ Record (rec :& name ::: sort) where
    modify (modRec :& name := mod) (rec :& _ := val) = modify modRec rec :& name := mod val
```

Figure 1. Definition of class *Record* with method *modify*

```
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)
```

Figure 2. Definition of class *Record* with method *fold*

It seems obvious to use emulation of type-level abstractions again to solve this problem. However, this does not work. In the type of *fold*, we would have to replace every type-level application of *thing* to some ρ by *App thing ρ*. As a result, *fold*'s type would contain *thing* only as an index of the type synonym family *App*. Since type synonym families are not necessarily injective, this would mean that whenever *fold* is used, the concrete substitute for *thing* could not be deduced.

Our solution is to introduce wrapper types that are isomorphic to the type-level functions we actually want to use. For every inductively defined combinator χ of a type $(Record\ rec) \Rightarrow \tau$, we introduce a type constructor Θ as follows:

newtype $\Theta\ rec = \Theta\ \tau$

Then we use *fold* to generate the wrapped combinator $\Theta\ \chi$. This is possible since that combinator has the type $(Record\ rec) \Rightarrow \Theta\ rec$, and Θ is a proper substitute for *thing*. Finally, we extract χ from $\Theta\ \chi$.

Figure 3 presents an implementation of *modify* that is based on *fold*. Note that the declarations of the functions *nilModify* and *snocModify* are basically the same as the declarations of *modify* in the two instance declarations of figure 1. Alas, the wrapping and unwrapping of combinators makes the new implementation more verbose. This kind of overhead is our reason to not use wrapper types for record styles.

5. Record Conversion

Our implementation of records imposes a total order on the fields of each record. While this is useful for distinguishing fields of the same name, it is undesirable otherwise. We want to be able to ignore such superfluous order. Furthermore, it is often beneficial if we can automatically ignore record fields we are not interested in. That way, record operations can be made more general since they can also work with records that contain more than the expected fields. Therefore, we introduce a conversion operator for records that is able to reorder and drop fields.

5.1 Equivalence and Convertibility

Let us look at the special case of records that do not contain multiple fields of the same name. Such records denote mappings from names to values. We call these mappings the meanings of the respective records and write $[\![r]\!]$ for the meaning of a record r. We say that two records r_1 and r_2 are equivalent, written $r_1 \approx r_2$, if and only if they have the same meaning. So two records are equivalent if they only differ in the order of fields. A record r can be converted into a record r', written $r \lesssim r'$, if and only if

$$\operatorname{dom}[\![r]\!] \supseteq \operatorname{dom}[\![r']\!] \wedge \forall\nu \in \operatorname{dom}[\![r']\!] : [\![r]\!](\nu) = [\![r']\!](\nu)\ .$$

So record conversion may reorder and drop fields arbitrarily. Note that $r_1 \approx r_2$ holds if and only if $r_1 \lesssim r_2 \wedge r_2 \lesssim r_1$.

Now, we want to also consider records that contain several fields of the same name. First, we modify the record semantics. The meaning of a record is now a function that maps each potential name, that is, each type of kind *, to the list of values that the record assigns to that name. The order of the values in the list matches the order of their respective fields in the record. Names that do not occur in the record are mapped to the empty list. Again, $r_1 \approx r_2$ shall hold if and only if $[\![r_1]\!] = [\![r_2]\!]$. So two records are equivalent if they contain the same fields and fields of the same name occur in the same order.

Having more values per name means that we cannot identify a field solely by its name anymore. We choose to identify a field of a record r by its name ν and the index of its value in $[\![r]\!](\nu)$. Thereby, we index the values in $[\![r]\!](\nu)$ backwards, so that the first element gets the largest and the last element gets the smallest index. This will make the implementation of record conversion easier. An ordinary front-to-back indexing would not play well with the fact that (:&) appends fields instead of prepending them. We can now reformulate the criterion for record equivalence. Two records are equivalent if and only if they contain the same fields and the fields have the same indices in both records.

We want to ensure that after a record conversion, fields are identified in the same way as they were identified before. So a field must keep its index during conversion. We define record convertibility such that $r \lesssim r'$ holds if and only if for each ν of kind *, $[\![r']\!](\nu)$ is a suffix of $[\![r]\!](\nu)$. Again, we have the fact that $r_1 \approx r_2$ is equivalent to $r_1 \lesssim r_2 \wedge r_2 \lesssim r_1$.

We can see a record scheme as a kind of record itself by treating sorts as values and types of the form $\nu ::: \varsigma$ as fields with name ν and value ς. That way, we can extend $[\![\cdot]\!]$, $\approx$, and $\lesssim$ to schemes.

$$modify :: (Record\ rec) \Rightarrow rec\ (ModStyle\ style) \rightarrow rec\ style \rightarrow rec\ style$$
$$modify = \textbf{let}$$
$$\qquad\qquad ModifyThing\ modify = fold\ modifyNilAlt\ modifySnocAlt$$
$$\qquad \textbf{in}\ modify$$
$$\textbf{newtype}\ ModifyThing\ style\ rec = ModifyThing\ (rec\ (ModStyle\ style) \rightarrow rec\ style \rightarrow rec\ style)$$
$$modifyNilAlt :: ModifyThing\ style\ X$$
$$modifyNilAlt = ModifyThing\ nilModify\ \textbf{where}$$
$$\quad nilModify\ X\ X = X$$
$$modifySnocAlt :: (Record\ rec) \Rightarrow ModifyThing\ style\ rec \rightarrow ModifyThing\ style\ (rec\ \text{:\&}\ name :::\ sort)$$
$$modifySnocAlt\ (ModifyThing\ modify) = ModifyThing\ snocModify\ \textbf{where}$$
$$\quad snocModify\ (modRec\ \text{:\&}\ name := mod)\ (rec\ \text{:\&}\ \_ := val) = modify\ modRec\ rec\ \text{:\&}\ name := mod\ val$$

Figure 3. Implementation of *modify* based on *fold*

5.2 Implementation of Record Conversion

Let ρ be a record scheme, σ be a record style and r be a record of type $\rho\ \sigma$. There is a bijection between the sets $\{\rho' \mid \rho \lesssim \rho'\}$ and $\{r' \mid r \lesssim r'\}$ such that for each concrete ρ' and corresponding r', r' has the type $\rho'\ \sigma$. The idea is that for each ρ', we can generate the corresponding r' from r by performing the same reorderings and droppings that we use to transform ρ into ρ'. So while a record r can usually be converted into different records r', we can select the desired conversion result via its scheme. We will use this in the implementation of record conversion.

We define a class *Convertible* with two parameters such that a pair of record schemes ρ and ρ' is an instance of *Convertible* if and only if $\rho \lesssim \rho'$. *Convertible* contains a method *convert* of type

$$(Convertible\ rec\ rec') \Rightarrow rec\ style \rightarrow rec'\ style\ .$$

This type implies that for each record r of type $\rho\ \sigma$, *convert* r has every type $\rho'\ \sigma$ for which $\rho \lesssim \rho'$ holds. For each concrete ρ', the type-restricted expression *convert* $r :: \rho'\ \sigma$ yields the conversion result r' that corresponds to ρ' according to the above-mentioned bijection.

Figure 4 shows the definition of *Convertible*. This definition uses induction on the scheme of the conversion result. It employs a helper class *Separation*. A quadruple of two record schemes ρ and ϱ, a name ν, and a sort ς is an instance of *Separation* if and only if the last (:::)-type in ρ that has name ν is $\nu ::: \varsigma$, and removing this type from ρ yields ϱ. The *separate* method extracts the last field of name ν from the given record and yields this field together with the remaining record.

Separation uses a functional dependency to specify that the scheme of the source record and the name of the extracted field uniquely determine the scheme of the remaining record. It seems more sensible to use a type synonym family to specify this dependency. After all, we already used the feature of type synonym families to implement emulation of type-level abstractions. The problem is that the instance declarations of *Separation* overlap, which would result in overlapping instance declarations for the type synonym family that we would introduce. However, overlapping is forbidden for type synonym families.

Now, let us look at the type equality constraint $sort \sim sepSort$ in the first instance declaration of *Separation*. This equality constraint ensures that the actual sort of the extracted field equals the specification of the extracted field's sort. Normally, we could eliminate this equality constraint by using a single type variable instead of the two different variables $sort$ and $sepSort$. That is, we could replace

$$(sort \sim sepSort) \Rightarrow$$
$$Separation\ (rec\ \text{:\&}\ name :::\ sort)\ rec\ name\ sepSort$$

by

$$Separation\ (rec\ \text{:\&}\ name :::\ sort)\ rec\ name\ sort\ .$$

However, because of our use of overlapping instances, this transformation would change the meaning of the program.

The original instance declaration head matches whenever the last name of the record scheme equals the name of the extracted field. If the last sort is not the one that is specified as the sort of the extracted field, the equality constraint is not fulfilled, and we get a type error. This is in line with our specification of *Separate* above.

If we would eliminate the equality constraint, the instance declaration head would not match in case the last name equals the name of the extracted field but the last sort is different from the required sort. However, the head of the second instance declaration would match in this case. Therefore, the *separate* method would not extract the last field that has the respective name but the last field whose name and sort are the required ones. This would be contrary to our specification of *Separation* and would lead to a bogus implementation of *Convertible*.

An advantage of the solution with the type equality constraint is that we only need a name to identify the field that we want to extract, not a sort. So the sort of the extracted field can be unknown initially and then determined by the equality constraint. This is useful, for example, in record pattern matching, which we will describe in subsection 5.4.

Support for type equality constraints was introduced into GHC as part of the type family extension since type equality constraints are often helpful when working with type synonym families. Our usage of equality constraints shows that they are also useful without type families. We therefore argue that they should be treated as a separate extension to the core language, independent of type families.

5.3 Modification without Limits

We will now discuss the function *modify* in its general form, that is, without the restriction we introduced at the beginning of section 3. We change the definition of *modify* from the introduction to accommodate records with multiple occurrences of the same name. In its original form, *modify* applies each function from the modification record to the value from the data record that has the same name as the function. Now, it shall apply each function from the modification record to the value of the data record that has the same name and index.

The *Convertible* class allows us to give the general *modify* function a type. This type is

$$(Record\ rec, Record\ modRec, Convertible\ rec\ modRec) \Rightarrow$$
$$modRec\ (ModStyle\ style) \rightarrow rec\ style \rightarrow rec\ style\ .$$

```
class Convertible rec rec' where
    convert :: rec style → rec' style
instance Convertible rec X where
    convert _ = X
instance (Separation rec remain name' sort', Convertible remain rec') ⇒
            Convertible rec (rec' :& name' ::: sort') where
    convert rec = convert remain :& field' where
        (remain, field') = separate rec
class Separation rec remain sepName sepSort | rec sepName → remain where
    separate :: rec style → (remain style, (sepName ::: sepSort) style)
instance (sort ∼ sepSort) ⇒
            Separation (rec :& name ::: sort) rec name sepSort where
    separate (rec :& field) = (rec, field)
instance (Separation rec remain sepName sepSort, (remain :& name ::: sort) ∼ extRemain) ⇒
            Separation (rec :& name ::: sort) extRemain sepName sepSort where
    separate (rec :& field) = (remain :& field, sepField) where
        (remain, sepField) = separate rec
```

Figure 4. Implementation of record conversion

The *convert* method makes it possible to implement *modify*. Since the implementation is rather complicated, we only sketch its idea here. The complete code can be found in the *records* package [4].

Say we apply *modify* to a modification record that has type μ (*ModStyle* σ) and a data record of type ρ σ. The class assertion *Convertible* ρ μ ensures that all records of scheme ρ can be converted to corresponding records of scheme μ. So it might be tempting to apply *convert* to the data record. However, this would not be of any help. We would lose all fields that do not have an associated field in the modification record, whereas we should actually keep these fields with their original values. So we pursue a different road.

We first use induction on ρ to generate a record of scheme ρ whose values are so-called update functions. An update function in a field with a sort ς has the type

$$(App \ \sigma \ \varsigma \to App \ \sigma \ \varsigma) \to (\rho \ \sigma \to \rho \ \sigma) \ .$$

Applying the update function to a function f and a record r yields a new record that differs from r only in the value of the field that corresponds to the field of the update function. The new value of that field is formed by applying f to its old value.

Next, we apply *convert* to the record of update functions to get an adjusted record of update functions that has scheme μ. Using induction on μ, we apply each update function to the corresponding function from the modification record and form the composition of the resulting functions. We apply this composition to the data record, which gives us the modified data record.

5.4 Record Pattern Matching

Since records are values of algebraic data types, we can use pattern matching to access the values of their fields. However, a pattern must be of the same type as the record that is matched against it. So the pattern must contain one subpattern for each record field, and these subpatterns must occur in the order their corresponding fields occur in. This is a major drawback in comparison to other record systems. We want to be able to reorder and drop fields automatically during pattern matching.

If we replace a record r with the record *convert* r, the type of the record is changed from ρ σ to (*Convertible* ρ rec') ⇒ rec' σ. We can specify the concrete scheme of the conversion result by matching *convert* r against a pattern of the form

$$X \ :\& \ \nu_1 := \pi_1 \ :\& \ \ldots :\& \ \nu_n := \pi_n \ \ ,$$

where ν_1 through ν_n are concrete names, and π_1 through π_n are arbitrary patterns. We do not need to assign sorts to the patterns π_i. The reason is that instance selection for *Convertible* and *Separate* only depends on names, and the π_i automatically get the correct sorts from ρ because of the equality constraint $sort \sim sepSort$.

6. First-Class Subkinds

The *fold* operator from section 4 can only generate combinators that work for all record schemes. However, there are record combinators that only work on record schemes whose sorts fulfill certain conditions.

As an example, let us look at a function that works with records of arrays. Haskell offers a binary type constructor *Array*. A type *Array* ι η, where ι is an instance of the *Ix* class, covers all arrays with indices of type ι and elements of type η. There is a function *elems* of type

$$(Ix \ ix) \Rightarrow Array \ ix \ el \to [el]$$

that converts an array into the list of its elements. Say we want to define a function *mapElems* that converts a record of arrays into a record of lists by repeatedly applying *elems*. It seems obvious to introduce two new data types *ArrayStyle* and *ElemsStyle* and give *mapElems* the type

$$(Record \ rec) \Rightarrow rec \ ArrayStyle \to rec \ ElemsStyle \ .$$

Each sort in the record scheme used by *mapElems* has to specify an index type and an element type. Therefore, such a sort must be essentially a pair of types instead of a single type. However, a pair of types ι and η can be represented by a single type Π ι η where Π is some fixed type constructor of kind $* \to * \to *$. We take *Array* as our Π so that the sorts are the array types. This leads to the following instance declarations for *App*:

```
type instance
    App ArrayStyle (Array ix el) = Array ix el
type instance
    App ElemsStyle (Array ix el) = [el]
```

Now, *mapElems* can only work with record schemes whose sorts are of the form $Array\ \iota\ \eta$ with ι being an instance of Ix. So the above type for *mapElems* is too general since it allows arbitrary record schemes. It is also not possible to implement *mapElems* via the *fold* operator as defined in section 4. This *fold* operator requires a (:&)-alternative which places no restrictions on the last sort of the record scheme. To see this, remember that the type of the second argument of *fold* is

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

However, the (:&)-alternative in the definition of *mapElems* has to apply *elems* to the value of the last field so that it has the less general type

$$\forall rec\ name\ ix\ el.(Record\ rec, Ix\ ix) \Rightarrow$$
$$thing\ rec \rightarrow thing\ (rec :\&\ name :::\ Array\ ix\ el)\ .$$

To solve this problem, we introduce the notion of subkind. Subkinds are the kind-level analog of subtypes. So a subkind of a kind ξ denotes a set of types that all have kind ξ. We refine the *Record* class such that it supports inductive definitions over all record schemes whose sorts have a given subkind of kind $*$. In the case of *mapElems*, we use the subkind of all proper array types. For functions that are defined on all record schemes, we use $*$ itself.

6.1 Emulation of Subkinds

We can emulate subkinds of a fixed base kind, whereby the base kind will be $*$ in our case. We represent subkinds by types. That way, subkinds are first-class citizens at the type level. In addition, we can use type polymorphism to emulate subkind polymorphism.

We introduce a two-parameter class *Inhabitant* with no methods. Each pair of a subkind representation and a type that has the respective subkind is an instance of *Inhabitant*. So we have the following declarations for the subkind of array types:

data $Kind_{Array}$

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

Now, let us give a more complex example. Haskell provides a type constructor *Map*. For types κ and υ where κ is an instance of *Ord*, $Map\ \kappa\ \upsilon$ is the type of all finite maps from keys of type κ to values of type υ. However, there is also the type *IntMap*, which offers a more efficient implementation of maps whose keys are of type *Int*. We want to form the subkind of map types, which shall cover both of the above variants. So we use two instance declarations:

data $Kind_{Map}$

instance $(Ord\ key) \Rightarrow$
$\qquad\qquad Inhabitant\ Kind_{Map}\ (Map\ key\ val)$

instance $Inhabitant\ Kind_{Map}\ (IntMap\ val)$

Finally, we show that we are also able to represent kind $*$, which is, of course, a subkind of itself:

data $Kind_{Star}$

instance $Inhabitant\ Kind_{Star}\ val$

Now that we have looked at some examples, let us discuss the general picture. Imagine we had a new language construct for declaring subkinds of kind $*$. The declaration

subkind $\Xi = \Gamma_1 \Rightarrow \tau_1\ |\ \ldots\ |\ \Gamma_n \Rightarrow \tau_n$

introduces a subkind Ξ. Thereby, Γ_1 through Γ_n are contexts, and τ_1 through τ_n are types. They have to satisfy the following conditions:

- $\mathrm{FV}(\Gamma_i) \subseteq \mathrm{FV}(\tau_i)$ for all i with $1 \leqslant i \leqslant n$.

- For all i and j with $1 \leqslant i < j \leqslant n$, the types τ_i and τ_j cannot be unified.

The declared subkind Ξ covers all types τ for which there is an i and a variable assignment σ such that $\tau = \sigma(\tau_i)$ and the context $\sigma(\Gamma_i)$ holds.

To emulate the above subkind declaration, we first introduce an empty data type $Kind_\Xi$:

data $Kind_\Xi$

Afterwards, we provide an instance declaration of the following form for every i with $1 \leqslant i \leqslant n$:

instance $\Gamma_i \Rightarrow Inhabitant\ Kind_\Xi\ \tau_i$

Of course, we can use subkind declarations to introduce the three example subkinds:

subkind $Array = (Ix\ ix) \Rightarrow Array\ ix\ el$

subkind $Map\quad = (Ord\ key) \Rightarrow Map\ key\ val$
$\qquad\qquad\qquad | \ IntMap\ val$

subkind $Star\quad = val$

If we transform these declarations into data type and instance declarations, we end up with the code from the beginning of this subsection.

We change the *Record* class such that we can specify a subkind that all sorts have to belong to. We add a parameter to *Record* such that the class assertion $Record\ \varkappa\ \rho$ means that ρ is a record scheme that contains only sorts of the subkind represented by $\varkappa$. The new definition of *Record* is shown in figure 5. It differs from the original one in the following points:

- All references to the *Record* class contain an additional parameter *kind*.

- The head of the second instance declaration contains an additional assertion *Inhabitant kind sort*, which enforces that sorts are of the specified subkind.

- The type of *fold*'s second argument contains an additional class assertion *Inhabitant kind sort*, so that (:&)-alternatives only have to work with schemes whose last sort has the respective subkind.

Having the new *Record* definition, we give *mapElems* the type

$$(Record\ Kind_{Array}\ rec) \Rightarrow$$
$$rec\ ArrayStyle \rightarrow rec\ ElemsStyle\ .$$

A problem with the new definition of *Record* is that the class parameter *kind* does not occur in the type of *fold* except in class assertions. So when *fold* is used in some expression, the actual *kind* parameter cannot be determined. This occurs, for example, in the declaration of *fold* for the (:&)-case. GHC complains that it cannot deduce the context $(Record\ kind_1\ rec)$ from the context

$$(Record \qquad kind_2\ (rec :\&\ name :::\ sort),$$
$$Record \qquad kind_2\ rec,$$
$$Inhabitant\ kind_2\ sort)\ .$$

The variable $kind_1$ denotes the *kind* parameter of the *fold* in the expression *snocAlt* (*fold nilAlt snocAlt*), while $kind_2$ denotes the *kind* parameter of the *fold* in the left-hand side. GHC cannot see why both parameters should be equal.

There is a second, more serious, problem. Since Haskell classes are open, we cannot prevent subkinds from being extended. For example, someone could import our definition of the *Array* subkind and add the type *Bool* to this subkind using the following instance declaration:

instance $Inhabitant\ Kind_{Array}\ Bool$

```
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
    fold nilAlt _ = nilAlt
instance (Record kind rec, Inhabitant kind sort) ⇒ Record kind (rec :& name ::: sort) where
    fold nilAlt snocAlt = snocAlt (fold nilAlt snocAlt)
```

Figure 5. Definition of class *Record* with kinded sorts

So we have no guarantee that $(Inhabitant\ Kind_{Array}\ \varsigma)$ holds only for those ς that are of the form $Array\ \iota\ \eta$ with $(Ix\ \iota)$. This makes it impossible to define $mapElems$ using $fold$. Say θ is the *thing* type of the inductive definition of $mapElems$. Then the $(:\&)$-alternative of this definition has the most general type

$$\forall rec\ name\ ix\ el.(Record\ Kind_{Array}\ rec, Ix\ ix) \Rightarrow$$
$$\theta\ rec \rightarrow \theta\ (rec :\&\ name ::: Array\ ix\ el)\ .$$

This type is less general than the required type

$$\forall rec\ name\ sort.(Record\qquad Kind_{Array}\ rec,$$
$$Inhabitant\ Kind_{Array}\ sort) \Rightarrow$$
$$\theta\ rec \rightarrow \theta\ (rec :\&\ name ::: sort)\ .$$

In the next subsection, we will present a technique for closing subkinds, which we will use to solve this problem. As a side effect, we will also get rid of the problem that the *kind* parameter of a *fold* application cannot be inferred.

6.2 Closing Subkinds

Before we discuss how to close subkinds in general, we look at some examples again. To close the *Array* subkind, we must ensure that the set of types ς with $(Inhabitant\ Kind_{Array}\ \varsigma)$ is the same as the set of types $Array\ \iota\ \eta$ with $(Ix\ \iota)$. We can enforce this by making sure that universal quantification over all ς with $(Inhabitant\ Kind_{Map}\ \varsigma)$ is the same as universal quantification over all array types. That is for any type-level function F, the types

$$\forall sort.(Inhabitant\ Kind_{Array}\ sort) \Rightarrow F\ sort$$

and

$$\forall ix\ el.(Ix\ ix) \Rightarrow F\ (Array\ ix\ el)$$

are isomorphic. If we set F to

$$\lambda array \rightarrow \forall rec\ name.(Record\ Kind_{Array}\ rec) \Rightarrow$$
$$\theta\ rec \rightarrow \theta\ (rec :\&\ name ::: array)\ ,$$

those types correspond to the required and inferred type of the $(:\&)$-alternative of the $mapElems$ definition. If these are isomorphic, the former cannot be more general than the latter anymore. This allows us to define $mapElems$ using $fold$.

To close the *Map* subkind, we have to make sure that universal quantification over all ς with $(Inhabitant\ Kind_{Map}\ \varsigma)$ is the same as universal quantification over all types of subkind *Map*. The question is how the latter can be expressed. After all, the types of subkind *Map* do not all share a common structure. On the one hand, we have the ordinary map types of the form $Map\ \kappa\ \upsilon$ with $(Ord\ \kappa)$, on the other hand, we have the types of the form $IntMap\ \upsilon$. However, we can universally quantify over only the ordinary map types and also over only the $IntMap$ types. We will now show how we can use this to universally quantify over all types of subkind *Map*.

A type $\forall \alpha :: \xi.\tau'$ is isomorphic to the dependent function type $(\alpha :: \xi) \rightarrow \tau'$, that is, the type of all functions that map each type τ

of kind ξ to a value of type $\tau'[\tau/\alpha]$. Say we split ξ into two non-overlapping subkinds ξ_1 and ξ_2. Then we can split each function of type $(\alpha :: \xi) \rightarrow \tau'$ into two functions of types $(\alpha :: \xi_1) \rightarrow \tau'$ and $(\alpha :: \xi_2) \rightarrow \tau'$, respectively. In addition, we can merge two such functions to get back the corresponding function of type $(\alpha :: \xi) \rightarrow \tau'$. So a type $\forall \alpha :: \xi.\tau'$ is isomorphic to the type

$$(\forall \alpha :: \xi_1.\tau', \forall \alpha :: \xi_2.\tau')\ .$$

Therefore, we can close the *Map* subkind by ensuring that there is an isomorphism between

$$\forall sort.(Inhabitant\ Kind_{Map}\ sort) \Rightarrow F\ sort$$

and

$$(\forall key\ val.(Ord\ key) \Rightarrow F\ (Map\ key\ val),$$
$$\forall val.\qquad\qquad F\ (IntMap\ val))$$

for every type-level function F.

It is now easy to see how we can close subkinds in general. Say Ξ is a subkind that is declared as follows:

subkind $\Xi = \Gamma_1 \Rightarrow \tau_1 \mid \ldots \mid \Gamma_n \Rightarrow \tau_n$

Then we have to make sure that for all type-level functions F,

$$\forall sort.(Inhabitant\ Kind_{\Xi}\ sort) \Rightarrow F\ sort$$

is isomorphic to

$$(\forall A_1.\Gamma_1 \Rightarrow F\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow F\ \tau_n)\ ,$$

where for each i with $1 \leqslant i \leqslant n$, A_i is a whitespace-separated sequence of the free variables of τ_i.

It is sufficient to enforce the existence of isomorphisms only for those type-level functions F that can be represented without using type-level abstractions, that is, for all types F of kind $* \rightarrow *$. The reason is that for any type-level function F, we can introduce a type Φ that is isomorphic to F as follows:

newtype $\Phi\ arg = \Phi\ (F\ arg)$

If f is an isomorphism for Φ, the function $\Phi^{-1} \circ f \circ \Phi$ is an isomorphism for F.

We do not allow different isomorphisms for different types F. Instead, we require a single isomorphism for all types F of kind $* \rightarrow *$. We do so by demanding the existence of two functions

$$\overrightarrow{f_{\Xi}} :: (\forall sort.(Inhabitant\ Kind_{\Xi}\ sort) \Rightarrow item\ sort) \rightarrow$$
$$(\forall A_1.\Gamma_1 \Rightarrow item\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow item\ \tau_n)$$

and

$$\overleftarrow{f_{\Xi}} :: (\forall A_1.\Gamma_1 \Rightarrow item\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow item\ \tau_n) \rightarrow$$
$$(\forall sort.(Inhabitant\ Kind_{\Xi}\ sort) \Rightarrow item\ sort)$$

with $\overrightarrow{f_{\Xi}} \circ \overleftarrow{f_{\Xi}} = \overleftarrow{f_{\Xi}} \circ \overrightarrow{f_{\Xi}} = id$. Note the use of the type variable *item* which can be specialized to each type of kind $* \rightarrow *$.

We will now show how we can actually enforce the existence of such functions $\overrightarrow{f_{\Xi}}$ and $\overleftarrow{f_{\Xi}}$. Let us first look at the functions $\overrightarrow{f_{\Xi}}$,

which perform "forward conversions". We introduce a type class $Kind$ of all subkind representations. $Kind$ contains a method $closed$ whose implementations perform the forward conversions of the respective subkinds. Furthermore, we change the definition of $Record$ such that the $kind$ parameter must be an instance of $Kind$. This ensures that if we use records with sorts of a certain subkind, there is a forward conversion for that subkind.

Say $kind$ is the type variable used in the head of the class declaration of $Kind$. Then the argument type of $closed$ is

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

The structure of the result type depends on the concrete subkind. So we cannot come up with a single result type that uses $kind$ only as an ordinary type parameter. Instead, we have to use $kind$ as a type index for selecting the particular result type. We introduce an associated data family [1] All for this purpose. For every subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$, the type $All\ Kind_\Xi$ is isomorphic to

$$\lambda item \rightarrow (\forall A_1.\Gamma_1 \Rightarrow item\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow item\ \tau_n)$$

where the A_i are defined as above.

The complete class declaration of $Kind$ is shown in figure 6. For each subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$, we make $Kind_\Xi$ an instance of $Kind$ using an instance declaration of the following form:

instance $Kind\ Kind_\Xi$ **where**
    data $All\ Kind_\Xi\ item = All_\Xi\ (\forall A_1.\Gamma_1 \Rightarrow item\ \tau_1)$
                    $\cdots$
                    $(\forall A_n.\Gamma_n \Rightarrow item\ \tau_n)$
    $closed\ item = All_\Xi\ item\ \ldots\ item$

The type of the argument of $closed$ is

$$\forall sort.(Inhabitant\ Kind_\Xi\ sort) \Rightarrow item\ sort\ .$$

On the right-hand side of the definition of $closed$, this type is specialized to the types $\forall A_i.\Gamma_i \Rightarrow item\ \tau_i$. These specializations are possible because we have introduced an instance declaration of the following form for each i:

instance $\Gamma_i \Rightarrow Inhabitant\ Kind_\Xi\ \tau_i$

The concrete instance declarations of $Kind$ for $Array$, Map, and $*$ are shown in figure 7.

Now, we will introduce a function that performs "backwards conversions". The type of this function is

$$(Kind\ kind) \Rightarrow$$
$$All\ kind\ item \qquad\qquad\qquad\qquad \rightarrow$$
$$(\forall sort.(Inhabitant\ kind\ sort) \Rightarrow item\ sort)\ .$$

A type $\tau \rightarrow (\forall \alpha.\Gamma \Rightarrow \tau')$ is equivalent to $\forall \alpha.\Gamma \Rightarrow \tau \rightarrow \tau'$ as long as α does not occur free in τ. So the backwards conversion function has also the type

$$(Kind\ kind, Inhabitant\ kind\ sort) \Rightarrow$$
$$All\ kind\ item \rightarrow item\ sort\ .$$

We add a context $(Kind\ kind)$ to the class declaration of $Inhabitant$ and declare the function for backwards conversion as a method of $Inhabitant$. The class declaration of $Inhabitant$ now looks as follows:

class $(Kind\ kind) \Rightarrow Inhabitant\ kind\ sort$ **where**
    $specialize :: All\ kind\ item \rightarrow item\ sort$

For a concrete subkind inhabitant ς, $specialize$ converts from types with universal quantification over all inhabitants to the corresponding types that fix the inhabitant to ς. That is where $specialize$ got its name from.

For each subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$ and each i with $1 \leqslant i \leqslant n$, we need an instance declaration of the following form:

instance $\Gamma_i \Rightarrow Inhabitant\ Kind_\Xi\ \tau_i$ **where**
    $specialize\ (All_\Xi\ \_^{i-1}\ item\ \_^{n-i}) = item$

Hereby, $\_^k$ stands for a whitespace-separated sequence of k wildcard patterns (_). Figure 8 shows the concrete instance declarations for our three example subkinds.

Of course, the class declarations of $Kind$ and $Inhabitant$ do not ensure that instance declarations are formed according to the rules described above. So there is no guarantee that $closed \circ specialize = specialize \circ closed = id$ holds in fact. However, this is a general problem with Haskell's class system. For example, sensible instance declarations of Ord have to fulfill the condition $(<) = flip\ (>)$ but the compiler cannot check whether they actually do.

Figure 9 shows the final definition of the $Record$ class. This definition forces $kind$ parameters to be instances of the $Kind$ class. In addition, the type of $fold$'s second argument now uses the All data family. Thus, $kind$ occurs in $fold$'s type not only in a context but also as a data family parameter. Therefore, actual $kind$ parameters can now be inferred. Note that this would not be possible if All would be a type synonym family since type synonym families are not guaranteed to be injective. The use of All makes the definition of a wrapper type $Expander$ necessary. For all θ, ρ, and ν, the type $Expander\ \theta\ \rho\ \nu$ is isomorphic to the type-level function

$$\lambda sort \rightarrow (\theta\ \rho \rightarrow \theta\ (\rho :\&\ \nu ::: sort))\ .$$

7. Related Work

Systems for extensible records appear either as language features or as libraries. We will discuss the former kind of record systems in the following subsection. Afterwards, we compare our system to HList, a library for heterogenous lists that covers a record system. Finally, we look at the concept of multiple occurrences of names.

7.1 Extensible Records as a Language Feature

There is a wide variety of proposals for language extensions that implement extensible records. Typically, such extensions do not allow for combinators that work on complete records. So there is no equivalent to our record scheme $fold$ or to record conversion. However, they provide operations for adding record fields as well as for selecting and removing fields by name. Adding record fields corresponds to our (:&)-operator, while field selection and removal is provided by the $Separation$ class.

A typical example of a system for extensible records is the one by Gaster and Jones [2], which was implemented in part as the Trex extension of the Haskell interpreter Hugs. Based on that system, Jones and Peyton Jones proposed a simpler one [7], which was intended to serve as a standard feature of future Haskell versions.

In both systems, a name may not occur more than once in a record. As a consequence, some functions require that a record lacks a certain name. An example of such a function is extension of a record with a field, where the record must not contain the name of the field. To enforce the absence of a name, both record systems use so-called "lacks" predicates, which may occur in contexts.

Furthermore, both systems do not feature an analog to record type families. However, the system by Gaster and Jones introduces two hardwired type-level functions to and $from$. Both take a type τ_0 and a record type[2] as arguments and return a record type. The

---

[2] Actually, this is not a record type but a so-called row. However, this detail is irrelevant here.

```
class Kind kind where
    data All kind :: (∗ → ∗) → ∗
    closed :: (∀sort.(Inhabitant kind sort) ⇒ item sort) → All kind item
```

Figure 6. Declaration of class $Kind$

```
instance Kind Kind_Array where
    data All Kind_Array item = All_Array (∀ix el.(Ix ix) ⇒ item (Array ix el))
    closed item = All_Array item
instance Kind Kind_Map where
    data All Kind_Map item = All_Map (∀key val.(Ord key) ⇒ item (Map key val))
                                     (∀val.                item (IntMap val))
    closed item = All_Map item item
instance Kind Kind_Star where
    data All Kind_Star item = All_Star (∀val.item val)
    closed item = All_Star item
```

Figure 7. $Kind$ instance declarations for $Array$, Map, and $\ast$

```
instance Ix ix ⇒ Inhabitant Kind_Array (Array ix el) where
    specialize (All_Array item) = item
instance Ord key ⇒ Inhabitant Kind_Map (Map key val) where
    specialize (All_Map item _) = item
instance Inhabitant Kind_Map (IntMap val) where
    specialize (All_Map _ item) = item
instance Inhabitant Kind_Star val where
    specialize (All_Star item) = item
```

Figure 8. $Inhabitant$ instance declarations for $Array$, Map, and $\ast$

```
class (Kind kind) ⇒ Record kind rec where
    fold :: thing X                                                          →
            (∀rec name.(Record kind rec) ⇒ All kind (Expander thing rec name)) →
            thing rec
newtype Expander thing rec name sort = Expander (thing rec → thing (rec :& name ::: sort))
instance (Kind kind) ⇒ Record kind X where
    fold nilAlt _ = nilAlt
instance (Record kind rec, Inhabitant kind sort) ⇒ Record kind (rec :& name ::: sort) where
    fold nilAlt expander = let
                    Expander snocAlt = specialize expander
                in snocAlt (fold nilAlt expander)
```

Figure 9. Final definition of class $Record$

to function replaces every field value type τ in the record type with $\tau \rightarrow \tau_0$, while *from* replaces every τ with $\tau_0 \rightarrow \tau$. The *from* function can be used, for example, to implement a generic introduction operator for records. Both *to* and *from* can be implemented in our system using record type families.

7.2 Extensible Records as a Library

HList [8] is a Haskell library for statically-typed heterogenous lists, that is, lists whose elements may have different types. HList uses heterogenous lists to represent records, which leads to a record implementation similar to the simple record library from section 2.

HList does not support record type families, folding of record schemes, or first-class subkinds. However, all record combinators that are implementable using these features can also be implemented in HList. A record combinator is implemented as the sole method of a dedicated type class that relates the different record types that appear in the type of the combinator. For example, one would introduce a class for the *modify* combinator as follows:

```
class (Record rec, Record modRec) ⇒
        Modify rec modRec where
    modify :: modRec → rec → rec
```

Here, $Record$ shall be the class of all record types, not schemes.

The downside of this approach is, that one gets many classes which are often related to each other but these relationships are usually not known to the type checker. For example, the type checker knows that each data record used by *modify* is a record but it does not know that each record can serve as a data record. This can result in large contexts that actually contain redundant information. This is similar to the problem that was mentioned in section 4 as the consequence of choosing option 1 to implement new record combinators.

HList also differs from our approach in its handling of field names. Field names are not represented by arbitrary types but essentially by type-level naturals. As a result, HList does not need overlapping instances to implement record conversion. Remember that we used overlapping instances only to select different instances depending on whether two names are equal or not. HList contains an equality check for type-level naturals that turns each pair of naturals into a corresponding type-level boolean. Such a boolean can be used to select the appropriate instance. The HList authors also implemented a general type equality check but they needed overlapping instances again to do so. Since we would use such an equality check only in one place, we decided to use overlapping instances directly in the implementation of record conversion.

Another difference regarding field names is that name types in HList contain no values apart from $\perp$. As a consequence, HList cannot provide pattern matching for records since there are no patterns that match individual names.

7.3 Scoped Names

Leijen [9] introduced the concept of records that may contain the same name multiple times. He points out that this yields a form of scoping over names. Leijen argues that scoped names are not only useful to make the life of the record system implementor easier but that they can lead to new applications of records.

8. Conclusions and Further Work

We have implemented a record system as a Haskell library, based solely on language features that are available today. Our record system covers the novel feature of record type families, a fold operator over record schemes, and a generic conversion operator for reordering and dropping fields. Together, these allow us to define a wide variety of generic record combinators that are statically typed. Furthermore, we have emulated first-class subkinds and subkind polymorphism in Haskell. This makes our record system even more powerful and might be also of interest outside record programming.

An open question is whether there are any performance issues involved in our implementation. After all, the linked-list implementation makes already simple field selection take linear time. Record conversion takes quadratic time since it contains two nested inductions. However, when record combinators are finally used in application code, their types are usually statically known. So it should be possible in principle to shift iteration over record schemes to compile time by using massive inlining. We still have to investigate whether GHC can do such inlining for us.

Implementing combinators using *fold* is not straightforward because values need to be wrapped and unwrapped. However, the task of writing all the necessary boilerplate code is rather mechanical. So it is likely that the boilerplate code can be generated by using Template Haskell [12], for example. Note, however, that using inductively defined combinators is easy.

A similar problem with verbose code occurs when emulating subkinds. So it might still be a good idea to provide subkind support as a language extension. Our record system would also profit from language support for names that would free us from explicitly declaring name types. However, note that our technique for pattern matching relies on names being represented by data constructors at the value level. Language-based name support should take this into account.

Existing proposals for record language support do not cover record type families, record scheme induction, and support for sorts of arbitrary subkinds. Since we have found these features to be very useful in practice, we argue that language support for records should not disallow them. It is probably best if the language provides only some basic support for record systems, and full record systems are then build on top of this as libraries.

Acknowledgments

References

[1] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, page 1–13, New York, NY, 2005. ACM.

[2] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[3] W. Jeltsch. *kinds-0.0.0.0* (Haskell package). `http://hackage.haskell.org/package/kinds-0.0.0.0`, Mar. 2010.

[4] W. Jeltsch. *records-0.0.0.1* (Haskell package). `http://hackage.haskell.org/package/records-0.0.0.1`, May 2010.

[5] W. Jeltsch. The Grapefruit homepage. `http://haskell.org/haskellwiki/Grapefruit`.

[6] W. Jeltsch. *type-functions-0.0.0.0* (Haskell package). `http://hackage.haskell.org/package/type-functions-0.0.0.0`, Apr. 2010.

[7] M. P. Jones and S. Peyton Jones. Lightweight extensible records for Haskell. In H. J. M. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28, Utrecht, 1999. Universiteit Utrecht.

[8] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, page 96–107, New York, NY, 2004. ACM.

[9] D. Leijen. Extensible records with scoped labels. In *Draft Proceedings of the 6th Symposium on Trends in Functional Programming*, page 297–312, Tallinn, Estonia, 2005. TTÜ Küberneetika Instituut.

[10] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, page 717–740, New York, NY, 1972. ACM.

[11] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, page 51–62, New York, NY, 2008. ACM.

[12] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, page 1–16, New York, NY, 2002. ACM.