Strongly Typed Heterogeneous Collections

August 26, 2004*

Oleg Kiselyov
FNMOC, Monterey, CA

Ralf Lämmel
VU & CWI, Amsterdam

Keean Schupke
Imperial College, London

Abstract

A heterogeneous collection is a datatype that is capable of storing data of different types, while providing operations for look-up, update, iteration, and others. There are various kinds of heterogeneous collections, differing in representation, invariants, and access operations. We describe HLIST — a Haskell library for strongly typed heterogeneous collections including extensible records. We illustrate HLIST's benefits in the context of type-safe database access in Haskell. The HLIST library relies on common extensions of Haskell 98. Our exploration raises interesting issues regarding Haskell's type system, in particular, avoidance of overlapping instances, and reification of type equality and type unification.

Categories and Subject Descriptors: E.2 [Data Storage Representations]; D.2.13 [Software Engineering]: Reusable Software; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms: Design, Languages.

Keywords: Collections, Extensible records, Type-safe database access, Dependently typed programming, Type-indexed rows, Type equality, Type improvement, Haskell.

1 Introduction

Programmers in typed functional languages are used to homogeneous collections, where values of the same type are stored in lists, sets, and others. There exist collection libraries, e.g., *Edison* for Haskell [26]. Homogeneous collections rely on parametric polymorphism. C++ programmers are also used to homogeneous collections such as those in the Standard Template Library, likewise for Ada and Eiffel. Java programmers are about to receive support for parametric polymorphism, finally. This may end the use of weakly typed collections ("Everything is of type `Object`!"), which require run-time type casts with the potential of unappreciated exceptions.

Unfortunately, the notion of typeful homogeneous collections fails to work for all the scenarios that require storing values of *different* types. Here is an open-ended list of typical examples that call for *heterogeneous collections*:

- A symbol table that is supposed to store entries of different types is heterogeneous. It is a finite map, where the result type depends on the argument value.
- An XML element is heterogeneously typed. In fact, XML elements are nested collections that are constrained by regular expressions and the 1-ambiguity property.
- Each row returned by an SQL query is a heterogeneous map from column names to cells. The result of a query is a homogeneous stream of heterogeneous rows.
- Adding an advanced object system to a functional language requires heterogeneous collections of a kind that combine extensible records with subtyping and an enumeration interface.

Weakly typed encodings are feasible for all the listed scenarios. For instance, a heterogeneously typed symbol table can be encoded using a suitably universal type, or dynamic typing, or type-safe cast.

The present paper introduces a strong typing discipline for heterogeneous collections. We deliver a dedicated Haskell library HLIST, which covers collection types such as lists, arrays, extensible records, type-indexed products and co-products. To this end, we advance techniques for dependently typed programming in Haskell [12, 21], and we rely on Haskell 98 with common extensions for multi-parameter classes and functional dependencies, as available in the GHC and Hugs implementations. (We manage to avoid overlapping instances — in the end!) Our development does not introduce yet another language extension, which is an improvement over earlier proposals for extensible records and other collection types [10, 31, 23, 29]. We explore some murky waters of Haskell's type system, such as the reification of type equality and type unification. While we have found portable, sound and practical ways around, more research is needed to deliver foundational clarifications that enable fundamental solutions. We identify the issues that need to be resolved.

The paper is structured as follows. In Sec. 2, we review weakly typed techniques for dealing with heterogeneous collections. In Sec. 3, we introduce *typeful heterogeneous lists*, which provide the basis for the HLIST library. We then work out different kinds of access operations and collection types:

- Sec. 4 — numeral-based access operations,
- Sec. 5 — labelled collections (or records),
- Sec. 6 — type-based access operations,
- Sec. 7 — type-indexed products.

In Sec. 8, we demonstrate the merits of heterogeneous collections in the context of type-safe database access in Haskell. In Sec. 9, we review our take on Haskell's type system. In Sec. 10, we discuss related work, and we conclude in Sec. 11. There are several appendices with various details. The source code from the paper can be retrieved from [1].

*A shorter version of this paper appeared in the proceedings of the ACM SIGPLAN Haskell Workshop 2004, September 22, 2004, Snowbird, Utah, USA, Published by ACM Press. This longer version provides several appendices and some extra paragraphs.

2 Not so strongly typed collections

We use database programming for the motivation of the HLIST library in this paper. We want to get to a point where SQL queries can be rephrased in Haskell in a typed and structured manner. As a simple example, let us attempt to encapsulate a simple SQL query in a Haskell function. The query should retrieve all animals (their keys and names) of a given breed from the 'foot-n-mouth' database. A query for sheep (rather than cows) looks as follows:

```
SELECT key,name FROM Animal WHERE breed = 'sheep';
```

Cheap strings

The following Haskell code encodes the parameterised query:

```
selectBreed :: String -> SqlHandle SqlQueryResult
selectBreed b =
  sqlQuery (  "SELECT key,name FROM Animal "
           ++ "WHERE breed ='" ++ b ++ "'")
```

Here we use a low-level ODBC binding for database access. The query is wrapped in an `SqlHandle` type, which encapsulates an IO action for an ODBC connection. The query function is parameterised in a `String` for the breed parameter. The type of query results is defined as follows:

```
type SqlQueryResult = ([ColName],[Row])
type ColName        = String
type Row            = [Cell]
type Cell           = String
```

That is, the result of a query consists of a list of column names and a list of rows, where a row in turn is a list of cells. Both column names and cells are plain strings. This is painful code in the eye of most programmers, but it is often a cheap way to make things work. Prominent database access techniques for all kinds of programming languages are string-based just like that.

Hand-made universes

If we wanted to maintain at least the primitive datatypes of cells, then we could replace the use of the string type with a universe of cell types (or a tagged union):

```
data Cell = IntObject     Int
          | FloatObject   Float
          | StringObject  String
          | ... -- and perhaps a few more cases
```

A row is still a list of such cells, but it is effectively a heterogeneous list. Instead of `Int` and `IntObject` we can use types and tags that are more descriptive of the columns, as below. Clearly, such an application-specific universe is subject to change whenever the data dictionary changes.

These are the types for the columns in the 'foot-n-mouth' database:

```
newtype Key   = Key Integer  deriving (Show,Eq,Ord)
newtype Name  = Name String  deriving (Show,Eq)
data Breed    = Cow | Sheep  deriving (Show,Eq)
newtype Price = Price Float  deriving (Show,Eq,Ord)
data Disease  = BSE | FM     deriving (Show,Eq)
...
```

We derive `Show`, `Eq`, and `Ord` instances to allow for printing of query results, and comparison of cells in `WHERE` conditions. We redefine `Cell` such that it is complete for the 'foot-n-mouth' database.

```
data Cell = KeyCell   Key
          | NameCell  Name
          | BreedCell Breed
          | ... -- and certainly more cases
```

The universal universe

Rather than introducing problem-specific universes of cell types, we can employ *dynamics* [2, 3]. Haskell's library `Data.Dynamic` provides the type `Dynamic` and an injection `toDyn` as well as a projection `fromDynamic`. Although this approach does not seem more typed, at least it is more extensible: we can make each new user-defined type amenable to injection and projection by providing an instance of Haskell's type class `Typeable`. There is a fully equivalent alternative: we can use existentially quantified cell types together with a nominal, extensible, type-safe cast [19].

Using dynamic typing, the encoding of column names in cell types allows us to leave them out in the type of query results. A row ends up being a heterogeneous list of `Dynamic`s. That is:

```
type SqlQueryResult = [Row]
type Row            = HList
type HList          = [Dynamic]
```

Using injection we construct an `HList`-typed value for cow Angus:

```
angus = [ toDyn (Key 42)
        , toDyn (Name "Angus")
        , toDyn Cow
        , toDyn (Price 75.5) ]
```

We can process such `HList`s with ordinary list-processing functions, e.g., `head`, `tail`, `null`, and `foldr`. We can also provide type-based operations, e.g., an operation `hOccursMany` to retrieve all elements of a given type:[1]

```
hOccursMany :: Typeable a => HList -> [a]
hOccursMany = map fromJust    -- unwrap Just
            . filter isJust   -- remove Nothing
            . map fromDynamic -- get out of Dynamic
```

For instance, we can attempt to look up the breed of cow Angus:

```
ghci-or-hugs> hOccursMany angus :: [Breed]
[Cow]
```

Note that printing `HList`s such as `angus` requires extra effort. A value of type `Dynamic` is normally opaque. We can revise `toDyn` to include a `Show` constraint in addition to the `Typeable` constraint. Alternatively, we can provide a `Show` instance for `Dynamic`, which attempts `fromDynamic` towards all showable types that we can possibly think of. These two options account for weak extensibility.

Too few or too many types

Most programmers are likely to loath operating on strings: it is completely untyped. A non-Haskell programmer might regard tagged unions as reasonably typed. The Haskell programmer will ask for much more typing. Most notably, the above type-based look-up gives no static guarantee that an element of the relevant type will be found at run-time.

In database programming, these guarantees correspond to static checks on column access in `WHERE` phrases and elsewhere. Static checks would require a mapping of the data dictionary to Haskell types. For example, we could define one `newtype` per database table, with each newtype describing table columns as a product or a record. We can process values of these newtypes with generic functions [19]. However, we are stuck: it is not enough to have precise types for database tables. We also would need precise types for queries and their intermediate expressions. So we face the need for an open-ended set of product or record types. This challenge is addressed below.

[1] To avoid confusion, we prefix all heterogeneously typed functions, types, and classes with an "h" (or, an "H") such as in `hOccursMany` and `HList`.

3 Typeful heterogeneous lists

We seek a notion of heterogeneous lists that is more typeful than `[Dynamic]`. The type of a list should precisely describe the types of its elements, as a type sequence or product. This will allow us to make static promises, e.g., a guarantee that a look-up operation for a type delivers a result. As we will see, precision of typing does not impair our ability to define 'normal' list-processing functionality.

Heterogeneous list constructors

We start by defining datatypes for lining up type sequences:

```
data HNil      = HNil       deriving (Eq,Show,Read)
data HCons e l = HCons e l  deriving (Eq,Show,Read)
```

These datatypes reify normal list structure at the type level, and thereby they allow us to statically distinguish empty and non-empty lists just as in dependently typed programming [12, 21]. Furthermore, each list element may have a different type.

For less parentheses, we assume right-associative infix operators:

```
type e :*: l = HCons e l  -- type level constructor
e .*. l      = HCons e l  -- value level constructor
```

Here is a type sequence for animals:

```
type Animal =
  Key :*: Name :*: Breed :*: Price :*: HNil
```

Here is a heterogeneous list that represents cow Angus:

```
angus :: Animal -- optional type declaration
angus =  Key 42
     .*. Name "Angus"
     .*. Cow
     .*. Price 75.5
     .*. HNil
```

We note that heterogeneous lists are essentially nested tuples. So we could use the normal type constructors `()` and `(,)` instead of `HNil` and `HCons` as in: `(Key,(Name,(Breed,(Price,()))))`. We favour fresh datatypes for building heterogeneous list. This helps avoiding confusion and clashes with 'normal' applications of `()` and `(,)`. We could also consider implicitly terminated type sequences. Again, we require a terminating `HNil` to avoid a mess.

A class of heterogeneous lists

When using `HCons` such as in `HCons e l`, we want the tail `l` to be a heterogeneous list type again. To this end, we will now work out a class `HList` whose extension is the set of all proper type sequences, i.e., the set of all nested, right-associative, binary products. This *class* replaces the *type* `[Dynamic]` from the previous section.

```
class HList l
instance HList HNil
instance HList l => HList (HCons e l)
```

What is the purpose of this class? Some readers might wonder whether we want to constrain the type constructor `HCons` like that:

```
data HList l => HCons e l = HCons e l deriving ...
```

After due discussion we decided: NO, being in good company [27]. The problem with constraints on datatypes is that they only imply a proof obligation, but type inference does not propagate them nicely. This would lead to a proliferation of `HList` constraints.

We rather place `HList` constraints on list-processing functionality whenever we want them. A user of the HLIST library does not employ the unconstrained constructor `HCons`, but only a constrained version of it. To this end, we retype `(.*.)`:

```
(.*.) :: HList l => e -> l -> HCons e l
(.*.) =  HCons
```

List-processing operations

Functions on normal lists (e.g., `head`, `tail`, and `null`) can be systematically transposed to the type level. Normally, each type-level operation is subject to a dedicated class; see App. A for some examples, and the HLIST source distribution for additional examples. Let us consider the recursive function for concatenation in some detail. For comparison, we recall normal list concatenation:

```
append :: [a] -> [a] -> [a]
append [] = id
append (x:l) = (:) x . append l
```

We define a class `HAppend` for concatenation of heterogeneous lists:

```
class HAppend l l' l'' | l l' -> l''
 where hAppend :: l -> l' -> l''
```

Here we use Haskell's extensions for multi-parameter classes and functional dependencies — which, incidentally, were introduced for the sake of 'normal' collection libraries. So it is not surprising that we end up using these extensions for heterogeneous collections. The functional dependency `l l' -> l''` indicates that the class is a type-level function — rather than a mere relation on types.

The instances follow the definition of `append` very closely:

```
instance HList l => HAppend HNil l l
 where hAppend HNil = id

instance (HList l, HAppend l l' l'')
      => HAppend (HCons x l) l' (HCons x l'')
 where   hAppend (HCons x l) = HCons x . hAppend l
```

We note that `append`'s equational term patterns show up twice in the class `HAppend`: once in the instance heads of `HAppend` and once in its method definitions. Also, the instance constraints for `HList` are like type checks to be performed at type checking 'run-time'. But otherwise we transcribe list processing to the heterogeneous case in a systematic manner. There is just a constant factor of noise.

Rather than defining all kinds of specific list-processing functions, one might wonder if the general recursion schemes for list processing can also be transcribed to the heterogeneous situation. This is indeed the case; see App. B for a heterogeneous fold operation and the HLIST source distribution for further higher-order operations on `HList`s.

Aside: stanamic lists

The class `HList`, and all the classes with list-processing operations (e.g., the shown `HAppend`) are in no way restricted to lists built from `HNil` and `HCons`. We can easily add instances for `HList`, `HAppend`, and others such that we also deal with less typeful heterogeneous lists (e.g., `[Dynamic]`), or with less generic heterogeneous lists (such as hand-made universes). This allows us to use our collection framework even in cases when the precise type sequence for a collection is not known statically, e.g., when collections are built from user input. One can even mix statically and dynamically typed collections. An advanced example of such a "stanamically" constrained data structure are the balanced trees in [18]. For the rest of the paper we will focus on statically typed heterogeneous lists.

4 Numeral-based access operations

We will now define array-like (or numeral-based) access operations for `HList`s. That is, we will use type-level naturals to address list elements. These access operations provide a basic layer in the `HList` library because type-based and label-based access operations can actually be implemented in terms of numeral-based operations.

```
class HNat n   => HLookupByHNat    n l e       |  n l   -> e         where  hLookupByHNat   :: n -> l -> e
class HNat n   => HDeleteAtHNat    n l l'      |  n l   -> l'        where  hDeleteAtHNat   :: n -> l -> l'
class HNat n   => HUpdateAtHNat    n e l l'    |  n e l -> l'        where  hUpdateAtHNat   :: n -> e -> l -> l'
class HNats ns => HProjectByHNats  ns l l'     |  ns l  -> l'       where  hProjectByHNats :: ns -> l -> l'
class HNats ns => HSplitByHNats    ns l l' l'' |  ns l  -> l' l''   where  hSplitByHNats   :: ns -> l -> (l',l'')
```

Figure 1. Numeral-based access operations for heterogeneous collections

Type-level naturals

Type-level naturals are represented by datatypes for zero and successor function. These datatypes are solely for the type-level: the only value of these types is $\perp$.[2]

```
class HNat n
data HZero; instance HNat HZero
data HSucc n; instance HNat n => HNat (HSucc n)
hZero :: HZero; hZero = ⊥
hSucc :: HNat n => n -> HSucc n; hSucc _ = ⊥
hPred :: HNat n => HSucc n -> n; hPred _ = ⊥
```

Eventually, one needs to perform all kinds of operations on type-level naturals such as arithmetics or comparison. As an example, we present (type-level) equality, as needed elsewhere in the paper.

First, we need type-level Booleans:

```
class HBool x
data HTrue; instance HBool HTrue
data HFalse; instance HBool HFalse
hTrue :: HTrue; hTrue = ⊥
hFalse :: HFalse; hFalse = ⊥
-- classes for HAnd and HOr omitted
```

Here are the classes for general type-level equality and comparison including the straightforward instances for the equality of naturals:

```
class HBool b => HEq x y b | x y -> b
class HBool b => HLt x y b | x y -> b
instance HEq HZero HZero HTrue
instance HNat n => HEq HZero (HSucc n) HFalse
instance HNat n => HEq (HSucc n) HZero HFalse
instance (HNat n, HNat n', HEq  n n' b)
      =>  HEq (HSucc n) (HSucc n') b
-- likewise for HLt
```

Induction on type-level naturals

One can define various access operations using naturals as indices; see Fig. 1 for an overview. For instance, the delete operation boils down to two instances: one for `HZero`; another for `HSucc`:

```
instance HDeleteAtHNat HZero (HCons e l) l
 where hDeleteAtHNat _ (HCons _ l) = l
instance (HDeleteAtHNat n l l', HNat n)
=> HDeleteAtHNat (HSucc n) (HCons e l) (HCons e l')
 where hDeleteAtHNat n (HCons e l)
     = HCons e (hDeleteAtHNat (hPred n) l)
```

Extra constraints

Functionality on collections carries *implied* constraints due to all the involved access operations. In addition, one might want to add *extra* constraints. For instance, we can use the following class to restrict the maximum length of a list (or an array):

```
class HMaxLength l s
instance (HLength l s', HLt s' (HSucc s) HTrue)
      =>  HMaxLength l s
```

[2] We also prefix all faked dependently typed functions and types with an "h" (or, an "H") such as in `hTrue` and `HBool`. These types correspond to subsets of values of normal types such as `Int`, and so let us discriminate the subsets of values at compile time.

```
class (HList l, HNat n) => HLength l n | l -> n
instance HLength HNil HZero
instance (HLength l n, HNat n, HList l)
      =>  HLength (HCons a l) (HSucc n)
```

By adding `HMaxLength` constraints to signatures or instances, one instructs Haskell to enforce size boundaries at compile time.

5 Extensible records

We will now define labelled collections, i.e., maps from labels to values. In essence, we will employ type-level naturals for labels, but we will enrich the structure of labels for convenience of programming with labelled collections. We end up defining extensible records this way, without requiring the language extensions of earlier proposals. From the point of view of database access, records provide the ultimate expressiveness for mapping column names to values in a typeful manner. Extensibility (and shrinkability) of records is key to dealing with types of joins and projections.

Haskell's nonextensible records recalled

In Haskell 98, we can define record types like this:

```
data Unpriced = Unpriced { key   :: Integer
                         , name  :: String
                         , breed :: Breed }
```

Here is a unpriced cow Angus:

```
unpricedAngus = Unpriced { key    = 42
                         , name   = "Angus"
                         , breed  = Cow }
```

What are access operations that are available for Haskell 98 records? We can retrieve components, and we can update records in a point-wise fashion:

```
ghci-or-hugs> breed unpricedAngus
Cow
ghci-or-hugs> unpricedAngus { breed = Sheep }
Unpriced{key=42,name="Angus",breed=Sheep}
```

We can not extend such records (unless we were thinking of nesting records and using polymorphic dummy fields for extension [6]). Also, we can not reuse labels among different record types, neither can we treat labels as data; so labels are not first-class citizens.

An extensible record demo

We place related labels in a namespace modelled by a silly datatype:

```
data FootNMouth = FootNMouth  -- a namespace
```

Labels in a namespace are constructed in a sequence starting with `firstLabel`, with `nextLabel` generating the next distinguished label. Each label is also annotated with a string for the label name. These are the labels for animals:

```
key   = firstLabel FootNMouth "key"
name  = nextLabel  key        "name"
breed = nextLabel  name       "breed"
price = nextLabel  breed      "price"
```

We build the record for the unpriced cow Angus as follows:

```
unpricedAngus =  key    .=. (42::Integer)
              .*. name   .=. "Angus"
              .*. breed  .=. Cow
              .*. emptyRecord
```

That is, record construction starts from `emptyRecord`; the label-value pairs are connected by "`.=.`"; and each label-value pair is added by using an overloaded operation "`.*.`".

Extensible records are printed more or less like Haskell 98 records:

```
ghci-or-hugs> unpricedAngus
Record{key=42,name="Angus",breed=Cow}
```

We retrieve a component from a record as follows:

```
ghci-or-hugs> unpricedAngus .!. breed
Cow
```

We can update components as follows:

```
ghci-or-hugs> unpricedAngus .@. breed .=. Sheep
Record{key=42,name="Angus",breed=Sheep}
```

We can really extend such records:

```
ghci-or-hugs> price .=. 8.8 .*. unpricedAngus
Record{price=8.8,key=42,name="Angus",breed=Cow}
```

One possible model of extensible records

Labels can be implemented by type-level naturals, qualified by a namespace, and annotated by a string for the label name:

```
data HNat x => Label x ns = Label x ns String
firstLabel = Label hZero
nextLabel (Label x ns _) = Label (hSucc x) ns
```

Records are maps from labels to values. We could go for heterogeneous lists of pairs; we could also go for pairs of heterogeneous lists of equal length. We abstract from this choice as follows:

```
class HZip x y l | x y -> l, l -> x y
 where hZip   :: x -> y -> l
       hUnzip :: l -> (x,y)
```

A record is a zipped list wrapped within `Record`:

```
newtype Record r = Record r -- to be constrained
```

Record construction is constrained as follows:

```
mkRecord :: (HZip ls vs r, HLabelSet ls)
         => r -> Record r
mkRecord = Record
```

For instance, the empty record is denoted as follows:

```
emptyRecord = mkRecord $ hZip HNil HNil
```

Labels in a record must be distinct:

```
class HLabelSet ls
instance HLabelSet HNil
instance ( HNat n, HMember (Label n ns) ls HFalse
         , HLabelSet ls )
      =>   HLabelSet (HCons (Label n ns) ls)
```

To this end, we define `HEq`-based membership test as follows:

```
class HBool b => HMember e l b | e l -> b
instance HMember e HNil HFalse
instance ( HEq e e' b     -- compare e and head e'
         , HMember e l b' -- use of label in tail
         , HOr b b' b''   -- type-level OR
         ) =>  HMember e (HCons e' l) b''
```

We also extend equality, which was already defined for type-level naturals, such that we can compute equality of labels. Here we assume that the labels in a record are in the same namespace:

```
instance HEq x x' b -- compare naturals in labels
      => HEq (Label x ns) (Label x' ns) b
```

Access operations

In the demo, we encountered access operations for look-up, update, and extension. There are also operations for appending records, for deletion of a label and its value in a record, for renaming of a label in a record, for projection and splitting of a record according to a label set. We can implement these operations directly on the representation of records (cf. "pair of lists" vs. "list of pairs"). Alternatively, we can use numeral-based access complemented by zipping and unzipping.

For instance, deletion ("`.-.`") can be defined as follows:

```
(Record r) .-. l = Record r'
   where (ls,vs) = hUnzip r
         n       = hFind l ls -- uses HEq on labels
         ls'     = hDeleteAtHNat n ls
         vs'     = hDeleteAtHNat n vs
         r'      = hZip ls' vs'
```

That is, we unzip the record; we find the index `n` of the given label `l` in the list `ls` of labels; we delete the subscripted elements in the lists `ls` and `vs` of labels and values; we finally re-zip the record.

6 Type-based access operations

Numeral-based and label-based access is in some sense still *value-based* — even though we had to reify naturals at the type level. We will now work out truly type-based access operations. From a database perspective, type-based operations are useful when types are descriptive of columns. In that case, there is no need to employ label-to-value mappings.

As for the coding style, we will make *transient* use of overlapping instances, as supported by the GHC and Hugs implementations of Haskell. We later circumvent overlapping instances.

Filter an `HList` for elements of a given type

The operation `hOccursMany` from Sec. 2 is an example of a type-based operation. The type of elements to be extracted from a list of dynamics is specified by fixing the result type of `hOccursMany`. We will now define such type-based operations on `HList` including more strongly typed ones; see Fig. 2 for an overview.

We dedicate a class to `hOccursMany`:

```
class HOccursMany e l where hOccursMany :: l -> [e]
```

The instance for `HNil` returns `[]`:

```
instance HOccursMany e HNil where hOccursMany _ = []
```

Another instance deals with a non-empty `HList` whose head is of the type of interest; notice that `e` is used twice in the instance head:

```
instance (HList l, HOccursMany e l)
      => HOccursMany e (HCons e l)
   where  hOccursMany (HCons e l) = e : hOccursMany l
```

There is yet another instance for a non-empty `HList` whose head is *not* of the same type as the element type in `hOccurs`'s result type:

```
instance (HList l, HOccursMany e l)
      => HOccursMany e (HCons e' l)
   where  hOccursMany (HCons _ l) = hOccursMany l
```

The two `HCons` instances are overlapping, while the former is more specific than the latter, which is thereby only applied when the former is not applicable, i.e., whenever the types `e` and `e'` are different.

`hOccursMany` is the regular "$*$" operation for type-based look-up. Then there are similar operations `hOccursMany1` (i.e., "$+$"), `hOccursOpt` (i.e., "?"), and `hOccursFst` (for the first occurrence). The class `HOccurs` and its complement `HOccursNot` require more thought. Most notably, a type-checked application of `hOccurs` is supposed to assure that there is exactly one element of the type in

```
class HOccursMany  e l   where  hOccursMany   ::  l -> [e]       -- return as many occurrences of type e as there are
class HOccursMany1 e l   where  hOccursMany1  ::  l -> (e,[e])   -- return at least one occurrence but all again
class HOccursOpt   e l   where  hOccursOpt    ::  l -> Maybe e   -- return the first occurrence if any
class HOccursFst   e l   where  hOccursFst    ::  l -> e         -- return the first occurrence out of one ore more
class HOccurs      e l   where  hOccurs       ::  l -> e         -- establish that there is precisely one occurrence
class HOccursNot   e l                                          -- constraint-only class for lack of occurrences
```

Figure 2. Type-based look-up operations for heterogeneous collections

question. Successful type checking of `hOccurs angus :: Breed` implies that `angus`'s breed is defined unambiguously. We will develop the definitions of `HOccurs` and `HOccursNot` in detail.

Documenting potential type errors

At first sight, there is no `HOccurs` instance for `HNil`, but we can provide one — be it for the sake of instructive error messages. Instances like the following make class-based dependently typed programming more manageable:

```
instance Fail (TypeNotFound e) => HOccurs e HNil
 where hOccurs = ⊥
```

Here we use a vacuous class `Fail` without instances, which just implements what its name promises, and we also assume a datatype `TypeNotFound` that serves for nothing but an error message:

```
class Fail x        -- no methods, no instances!
data TypeNotFound e -- no values, no operations!
```

Hence we obtain somewhat suggestive error messages:

```
ghci-or-hugs> hOccurs (HCons True HNil) :: Int
No instance for (Fail (TypeNotFound Int))
```

So we try to look up a value of a type that's not in the list. Hence, iteration ends up at `HNil`, and `TypeNotFound` is reported. Such documentary failure instances are used throughout the `HList` library.

Static look-up

We will now provide the actual definition of `hOccurs`. There are again two overlapping instances for non-empty lists; one for the case that the head fits with the type of interest, and another for recursion in case we haven't found an occurrence yet:

```
instance (HList l, HOccursNot e l)
      => HOccurs e (HCons e l)
  where  hOccurs (HCons e _) = e
instance (HList l, HOccurs e l)
      => HOccurs e (HCons e' l)
  where  hOccurs (HCons _ l) = hOccurs l
```

The constraint `HOccursNot e l` in the first instance assures that no elements of type `e` occur in the tail `l`. The class `HOccursNot` is for constraining only rather than actual look-up. Consequently, its definition does not comprise any method:

```
class HOccursNot e l    -- no methods!
data TypeFound e        -- for a failure instance
instance HOccursNot e HNil
instance (HList l, HOccursNot e l)
      => HOccursNot e (HCons e' l)
instance Fail (TypeFound e)
      => HOccursNot e (HCons e l)
```

The instances fold over `l` to test that each type is different from `e`. The last instance leads to failure for an offending head. This failure instance is obligatory because the more general instance for `HCons` would otherwise silently skip over the offending occurrence. Notice that Haskell's instance selection is solely based on syntactical matching. Hence, the failure of the more specific instance (via `Fail`) will *not* lead to reconsideration of the more general instance.

From look-up to projection

We can now readily define projection by mapping over a list of requested element types using simple look-up for each element type; see the HLIST source distribution for the actual code. For instance, the following query retrieves the key and the name of cow Angus:

```
ghci-or-hugs> hProject angus
              :: (HCons Key (HCons Name HNil))
HCons (Key 42) (HCons (Name "Angus") HNil)
```

This operation resembles projection in the sense of relational algebra, or in the sense of SQL's `SELECT` statements. (Think of the column names following the keyword `SELECT`.)

Type-based mutation operations

We also need mutation operations such as the following:

- Delete list elements identified by their type.
- Update list elements by values of the same type.
- Split a list into a projected list and its complement.

The update operation(s) mutate at the value level only, e.g.:

```
-- Replace the occurrences of type e
class  HUpdateMany  e l
 where hUpdateMany :: e -> l -> l
```

So the type-level programming bits of look-up can be adopted for type-preserving update. Deletion requires functional dependencies:

```
-- Delete the occurrences of type e in l, return l'
class  HDeleteMany e l l' | e l -> l'
 where hDeleteMany :: ... -- to be completed
```

Such mutation operations also mutate types. Without functional dependencies, users had to specify the result type explicitly, which is impractical. The trouble is that the combination of overlapping instances and functional dependencies leads us into murky water. We take this as an incentive to identify an overlapping-free idiom.

Passing on types as proxies

Let us first get the type of `hDeleteMany` right. It could be this one:

```
class  HDeleteMany e l l' | e l -> l'
 where hDeleteMany :: e -> l -> l'
```

The argument of type `e` would merely describe the type of the elements that should be deleted. We might not have any suitable value around (except $\perp$). Also, the above type obscures the role of the first argument. So we go for this type instead:

```
hDeleteMany :: Proxy e -> l -> l'
```

Proxies are defined as follow:

```
data Proxy e; proxy :: Proxy e; proxy = ⊥
```

Hence, the only value of a proxy type is the specific value $\perp$ of the constructed proxy type — not to be confused with the value $\perp$of the type being proxied. We can reduce values to proxies if needed:

```
toProxy :: e -> Proxy e; toProxy _ = ⊥
```

For example, we delete the name of cow Angus as follows:

```
ghci-or-hugs> hDeleteMany (proxy::Proxy Name) angus
HCons (Key 42) (HCons Cow (HCons (Price 75.5) HNil))
```

A non-solution

Adopting the style that we offered for look-up operations, we would want to implement `hDeleteMany` with one instance for `HNil`; one instance for 'delete head'; one instance for 'keep head':

```
instance HDeleteMany e HNil HNil
  where  hDeleteMany _ HNil = HNil

instance (HList l, HDeleteMany e l l')
      => HDeleteMany e (HCons e l) l'
 where   hDeleteMany p (HCons _ l) = hDeleteMany p l

instance (HList l, HDeleteMany e l l')
  => HDeleteMany e (HCons e' l) (HCons e' l')
 where   hDeleteMany p (HCons e' l)
       = HCons e' (hDeleteMany p l)
```

Alas, the two overlapping instance heads for `HCons` are in no substitution ordering. (Neither GHC nor Hugs can be persuaded to accept this code.)

Move patterns from the head to constraints

There is a rescue. We simply need to generalise one instance head so that it becomes more general than the other. Then, instance selection will be re-enabled. We generalise the head of the last instance:

- before: `HDeleteMany e (HCons e' l) (HCons e' l')`
- after: `HDeleteMany e (HCons e' l) l''`

But we must maintain the type equation `l''` equals `HCons e' l'`! To this end, we employ type cast. We add an instance constraint `TypeCast (HCons e' l') l''`, and we also cast in the method:

```
instance ( HList l, HDeleteMany e l l'
         , TypeCast (HCons e' l') l'' )
      =>   HDeleteMany e (HCons e' l) l''
 where     hDeleteMany p (HCons e' l)
         = typeCast (HCons e' (hDeleteMany p l))
```

There is no shortage of type-safe casts for Haskell [34, 8, 4, 19]. The one we need here is really resolved at the type-level. So there is no `Maybe` involved, since `typeCast` cannot fail at run-time:

```
class TypeCast x y | x -> y, y -> x
where typeCast :: x -> y
```

The functional dependencies capture our expectation of type cast to be an isomorphism on types (in fact, the identity function). We will discuss the implementation of `TypeCast` in Sec. 9.

Ended up in murky water

There is no real consensus on the overlapping instance mechanism as soon as functional dependencies are involved. Our result from above fits with GHC's model, but Hugs reports that the instances are inconsistent with the functional dependency for `HDeleteMany`. Here is a simple example that exercises this disagreement:

```
data Foo x y
class Bar x y | x -> y
class Zoo x y | x -> y
instance Zoo y r => Bar (Foo x y) r
instance Zoo z r => Bar (Foo (Foo x y) z) r
```

Hugs' type system misses the point that `Bar`'s second parameter is still functionally dependent on part of `Bar`'s first parameter.

Overlapping banned

We give up on persuading Hugs. Also, we do not want to depend on the doubtful future of overlapping instances in general. Furthermore, regimes for instance selection differ in ways other than consistency criteria for functional dependencies. For instance, GHC's instance selection is lazy, whereas Hugs' is eager.

We avoid overlapping instances by reformulating our problem into a case selection driven by a type-level Boolean denoting a computed type equality. The predicate for type equality is provided as follows:

```
class HBool b => TypeEq x y b | x y -> b
proxyEq :: TypeEq t t' b => Proxy t -> Proxy t' -> b
proxyEq _ _ = ⊥
```

We take for granted that we can define type equality; see Sec. 9. Using type equality, we replace the overlapping instances for `HDeleteMany` by the following case-preparing instance:

```
instance ( HList l, TypeEq e e' b
         , HDeleteManyCase b e e' l l' )
      =>   HDeleteMany e (HCons e' l) l'
 where     hDeleteMany p (HCons e' l)
  = hDeleteManyCase (proxyEq p (toProxy e')) p e' l
```

That is, we compute type equality so that we are able to decide whether the head needs to be deleted. This decision is then implemented by the helper class `HDeleteManyCase` with instances (i.e., branches) for the two Booleans:

```
class  HDeleteManyCase b e e' l l' | b e e' l -> l'
 where
  hDeleteManyCase :: b -> Proxy e -> e' -> l -> l'

instance HDeleteMany e l l'
      => HDeleteManyCase HTrue e e l l'
 where hDeleteManyCase _ p _ l = hDeleteMany p l

instance HDeleteMany e l l'
      => HDeleteManyCase HFalse e e' l (HCons e' l')
 where hDeleteManyCase _ p e' l
     = HCons e' (hDeleteMany p l)
```

This idiom works equally well for other type-based operations.

Type-to-natural mapping

We can even factor out case discriminations for type equality to be used in just a single location, namely in a type-to-natural mapping. The remaining type-based access operations can then employ this mapping completed by numeral-based access.

The type-to-natural mapping is hosted by the following class:

```
class HNat n => HType2HNat e l n | e l -> n
```

The implementation adopts the overlapping-free idiom:

```
instance (TypeEq e' e b, HType2HNatCase b e l n)
      =>  HType2HNat e (HCons e' l) n

class (HBool b, HNat n)
   =>  HType2HNatCase b e l n | b e l -> n
instance HOccursNot e l
      => HType2HNatCase HTrue e l HZero
instance HType2HNat e l n
      => HType2HNatCase HFalse e l (HSucc n)
```

We note that the first instance carries a constraint `HOccursNot e l`. This makes sure that the type `e` in question is associated with a single natural as index. Alternatively, we could return a list of a indexes for elements of type `e`. This would be necessary for the reconstruction of operations like `hOccursMany`.

For instance, type-based delete can now be expressed concisely in terms of numeral-based delete — without the hassle of a helper class for case discrimination on Booleans:

```
hDelete p l = hDeleteAtHNat (hType2HNat p l) l
```

Here we invoke the type-to-natural mapping using this function:

```
hType2HNat :: HType2HNat e l n => Proxy e -> l -> n
hType2HNat _ _ = ⊥
```

Aside: type schemas and class-based programming

The fine details of our heterogeneous collections reflect the employment of Haskell's class concept. Most notably, all involved type schemas must be sufficiently instantiated to allow for instance selection without causing ambiguities. This is just the same as in the case of `show . read` whose application to a string cannot be evaluated because the type of the intermediate result is not fixed.

We can store and look up polymorphic values as long as their type schemas are not needed for instance selection. So numeral-based access works fine even for arbitrary polymorphic elements, because the element types do not drive instance selection:

```
ghci-or-hugs> hLookupByHNat hZero (id .*. HNil) $ 42
42
```

The following type-based access still works:

```
ghci-or-hugs> hOccursMany (id .*. HNil) :: [Bool]
[]
```

We note that `hOccursMany` compares its result type with all element types. The type schema `forall a. a -> a` of `id` is sufficiently instantiated for that, i.e., `forall a. a -> a` is different from `Bool` for all possible `a`. Here is an example of an ambiguous situation:

```
ghci-or-hugs> hOccursMany (⊥ .*. HNil) :: [Bool]
No instance for ... <snipped>
```

The interaction of polymorphic elements in collections and class-based programming will continue to be a topic in the next section.

7 Type-indexed products

As a refinement of type-based access to heterogeneous collections, one can even require that a given collection is entirely *type-indexed*, i.e., that no type occurs more than once. Imposing this requirement on lists, we obtain so-called type-indexed products (TIPs; [31]). We will now briefly describe an implementation of TIPs. The dual of TIPs, TICs, are defined in App. C.

We wrap TIPs in a newtype so that we make the status of being type-indexed explicit in type signatures. Also, we can provide special instances for TIPs once we made this type distinction:

```
newtype TIP l  = TIP l -- to be constrained
unTIP  (TIP l) = l
```

The public constructor for TIPs supplies the key constraint for TIPs:

```
mkTIP :: HTypeIndexed l => l -> TIP l
mkTIP = TIP
```

The class `HTypeIndexed` is defined as follows:

```
class HList l => HTypeIndexed l
instance HTypeIndexed HNil
instance (HOccursNot e l,HTypeIndexed l)
      => HTypeIndexed (HCons e l)
```

The instances traverse over the type sequence, and the class `HOccursNot` is employed to assure that the type of the head does not occur (again) in the tail.

Let us upgrade `angus` to a TIP:

```
ghci-or-hugs> let myTipyCow = TIP angus
```

Lifting operations

Most trivially, there is a replacement for `HNil`:

```
emptyTIP = mkTIP HNil
```

Operations on TIPs are lifted as follows. "`TIP`" is unwrapped in arguments, and it is wrapped in the result (if this is a TIP), while constraints are added so that the `HTypeIndexed` property is enforced. For instance:

```
instance (HAppend l l' l'', HTypeIndexed l'')
      =>  HAppend (TIP l) (TIP l') (TIP l'')
where hAppend (TIP l) (TIP l') = mkTIP (hAppend l l')
```

Likewise we overload `(.*.)` to work for TIPs, i.e., extensions are assured to preserve the TIP property. To illustrate extension, we label `myTipyCow` with BSE:

```
ghci-or-hugs> BSE .*. myTipyCow
TIP (HCons BSE ...)
```

The animal `myTipyCow` is a cow; so it can't be a sheep then:

```
ghci-or-hugs> Sheep .*. myTipyCow
No instance for (Fail (TypeFound Breed))
```

Subtype constraints

TIPs naturally give rise to a subtype property. One TIP type l is a subtype of another TIP type l' if l contains all types from l'. This is expressed as follows:

```
class SubType l l'
instance SubType (TIP l) (TIP HNil)
instance (HOccurs e l, SubType (TIP l) (TIP l'))
      =>  SubType (TIP l) (TIP (HCons e l'))
```

From this it is clear that we do not care about the order of elements in the type-indexed products. We also note that the intersection of `HSubType` *x y* and `HSubType` *y x* immediately provides a faithful form of type equivalence for TIPs (while mere equality of the underlying type sequences would not be faithful).

As an aside, we can also instantiate subtyping for records. (This can be used in deriving an effective object system in Haskell.) A record type r is a subtype of some record type r' if r contains at least the labels of r', and the component types for the shared labels are the same. Projection according to label sets is of use here:

```
instance ( HZip ls vs r'
     , HProjectByLabels ls (Record r) (Record r') )
 =>    SubType (Record r) (Record r')
```

An idiom for constraint annotation

Let us review idiomatic support for adding extra constraints. For instance, let us deploy a constrained `hOccurs` that is meant to return the `Key` of an animalish TIP. TIPs that are not of a subtype of `TIP Animal` are to be rejected — even if they carry a `Key`. This can be encoded as follows:

```
animalKey :: ( SubType l (TIP Animal) -- extra
             , HOccurs Key l          -- implied
             ) => l -> Key
animalKey = hOccurs
```

The trouble is that this conservative approach forces one to gather all the implied constraints and to make them explicit just as the extra constraints. There is an idiom that allows one to solely enumerate extra constraints. Essentially, one defines a constrained identity function that imposes the constraints of interest on its argument.

The following identity function insists on animals:

```
animalish :: SubType l (TIP Animal) => l -> l
animalish = id
```

We can now discipline the `Key` getter as follows:

```
animalKey l = hOccurs (animalish l) :: Key
```

The subtype constraint takes action as one can see here:

```
ghci-or-hugs> animalKey myTipyCow
Key 42
ghci-or-hugs> animalKey (Key 42 .*. emptyTIP)
No instances for (Fail (TypeNotFound Price),
                  Fail (TypeNotFound Breed),
                  Fail (TypeNotFound Name))
```

The error message lists the types that are missing from `Animal`.

A polymorphism benchmark

As proposed by a reviewer of this paper, we will now consider an example from [31], which is, in a way, about type-based matching.

The following function selects two elements from a collection:

```
tuple l = let x  = hOccurs l
              l' = hDeleteAtProxy (toProxy x) l
              y  = hOccurs l'
          in (x,y)
```

The following session shows that we can match the elements of a collection in whatever order, while the overloaded operations in `tuple` are resolved by the consumers of the matched values:

```
ghci-or-hugs> let one = (1::Int)
ghci-or-hugs> let inc x = x + one
ghci-or-hugs> let incNot (a,b) = (inc a,not b)
ghci-or-hugs> let notInc (a,b) = (not b,inc a)
ghci-or-hugs> let oneTrue = one .*. True .*. HNil
ghci-or-hugs> incNot (tuple oneTrue)
(2,False)
ghci-or-hugs> notInc (tuple oneTrue)
(False,2)
```

The following example should arguably work, but it doesn't:

```
ghci-or-hugs> inc $ fst (tuple oneTrue)
No instances for ... <snipped>
```

We are going to make this work as well! We note that `oneTrue` stores two components; so by fixing the type of one component to `Int`, it should not matter that the type of the other component is left unspecified. The problem boils down to the following issue:

```
ghci-or-hugs> hOccurs (HCons True HNil)
No instance for (HOccurs e (HCons Bool HNil))
```

We would like to default `e` to `Bool` here. Rather than comparing the type of the head with a not yet instantiated result type, the two types should be unified. The `hOccurs` operation for TIPs does this:

```
ghci-or-hugs> hOccurs (True .*. emptyTIP)
True
ghci-or-hugs> let oneTrue = one .*. True .*. emptyTIP
ghci-or-hugs> inc $ fst (tuple oneTrue)
2
```

Even the following added polymorphism is handled:

```
ghci-or-hugs> let oneNull = one .*. [] .*. emptyTIP
ghci-or-hugs> inc $ fst (tuple oneNull)
2
```

The key idea is to provide a special instance for singleton lists, and to replace the test for type equality by unification via type cast:

```
instance TypeCast e' e
      => HOccurs e (TIP (HCons e' HNil))
where    hOccurs (TIP (HCons e' _)) = typeCast e'
instance HOccurs e (HCons x (HCons y l))
      => HOccurs e (TIP (HCons x (HCons y l)))
where hOccurs (TIP l) = hOccurs l
```

This example reveals that type cast provides a powerful idiom for *type improvement* — a more fine-grained one than functional dependencies. That is, type cast operates at the instance level as opposed to the class level!

8 Database programming

We will now demonstrate heterogeneous collections for database programming in Haskell. To this end, we adopt concepts from Leijen and Meijer's embedding approach for SQL [20]. We employ extensible records for two purposes:

- to represent the results of queries, and
- to represent schemas for relational algebra operations.

A detailed discussion of the approach is beyond the scope of this paper. We note however that the approach scales to the full relational algebra, and to a rich set of SQL idioms including all kinds of joins, existential quantification, nested queries, and table statements.

We recall the simple query from the beginning of the paper:

```
SELECT key,name FROM Animal WHERE breed = 'sheep';
```

In Haskell, we can now write this query in a type-safe manner.

```
selectBreed b = -- argument b for the breed
 do r1 <- table animalTable
    r2 <- restrict r1 (\r -> r .!. breed `SQL.eq` b)
    r3 <- project r2 (key .*. name .*. HNil)
    doSelect r3
```

Type inference works fine, but here is the type of the query anyway:

```
selectBreed :: Breed -> Query [
   Tkey   :=: AnimalId :*:
   Tname  :=: String   :*: HNil ]
```

That is, the result is a query for records with two components. (The types for the labels `key` and `name` are denoted by `Tkey` and `Tname`.)

The above do sequence encodes the SQL query in four steps:

- r1: We identify the table as in "`FROM Animal`".
- r2: We restrict the table according to the `WHERE` condition.
- r3: We perform projection as in "`SELECT key,name`".
- `doSelect r3`: The actual query is issued.

Steps 1–3 do *not* involve any database access. (Monadic style is used for hygienic name supply.) The operations `table`, `restrict`, `project` create or modify *type-annotated, syntactical expressions* for relations. The underlying key data structure looks as follows:

```
data Relation schema  -- type annotation layer
  =  Relation schema SqlRelation

data SqlRelation      -- expression layer
  =  SqlRelation {
   rTag           :: SqlTag,
   rSource        :: SqlSource,
   rRestrictList  :: [SqlExpression],
   rProjectList   :: [SqlExpression],
   rGroupList     :: [SqlExpression],
   rOrderList     :: [SqlExpression] }
```

That is, relations carry a schema, and their structural ingredients comprise a unique tag, a source (i.e., a database table), as well as lists of expressions describing restrictions (cf. `WHERE`), projections (including computed columns), grouping and ordering.

The type of the relational schema for animals is the following:

```
type AnimalSchema =
  Tkey   :=: Attribute AnimalId SqlInteger :*:
  Tname  :=: Attribute String   SqlVarchar :*:
  Tbreed :=: Attribute Breed    SqlVarchar :*:
  Tprice :=: Attribute Float    SqlNumeric :*:
  Tfarm  :=: Attribute FarmId   SqlInteger :*: HNil
```

The schema type lists both the domain of a column and the corresponding SQL type. For instance, the Haskell type for the `key` component is the newtype `AnimalId` rather than the SQL type `SqlInteger`. This 'domain as `newtypes`' technique increases type safety: one cannot possibly confuse an `AnimalId` and a `FarmId`. We note that some of the column types could be wrapped in `Maybe`, but this is not the case for `AnimalSchema`.

The datatype `Attribute` is a phantom type in its two type parameters. These phantoms drive coercions and make attribute access type-safe. For instance, consider the subexpression

`r .!. breed 'SQL.eq' b` for restriction in the above query. The look-up `r .!. breed` does not just establish that there is a `breed` component, but it also delivers a phantom-typed attribute, so that its use in the compound expression is type-constrained.

Structurally, attributes keep track of some details such as precision, and `NULL` constraints. All such information is extracted from the data dictionary of a database.

Here is a snippet of the extracted table description for animals:

```
animalTable :: Table AnimalSchema
animalTable =  mkTable "Animal" (
 key  .=. Attribute { ... } .*.
 name .=. Attribute { ... } .*.
 ... HNil )
```

This is all what's needed to make attribute access type-safe. Returning typed query results relies on further provisions. That is, the action `doSelect` for executing a query has to recast query results such that they are phrased in the Haskell types for column domains.

The code for the execution of `SELECT`s makes it all clear:

```
doSelect (Relation schema rel) = do
   sqlDo (showSqlRelation rel)
   rows <- getSqlRows
   return $ map ( labelHList labels
                . readHList  values
                ) rows
  where (labels,values) = hUnzip schema
```

The subexpression `showSqlRelation rel` computes the `SELECT` statement as a string, which is then given to `sqlDo` — the low-level, ODBC-based SQL handler. In the next step, we get all the queried rows as a lazy list of lists using this SQL service:

```
getSqlRows :: SqlHandle [[Maybe String]]
```

The subsequent `map` transforms the string-based `rows` into typeful ones in two steps. Firstly, we build an `HList` from the strings with `readHList`, while we use the attributes from the schema to drive this heterogeneous list construction. Secondly, we turn the `HList` into a record, while we reuse the labels of the schema.

9 By chance or by design?

We will now discuss the issues surrounding the definition of type equality, inequality, and unification — and give implementations differing in simplicity, genericity, and portability.

We define the class `TypeEq x y b` for type equality. The class relates two types `x` and `y` to the type `HTrue` in case the two types are equal; otherwise, the types are related to `HFalse`. We should point out however groundness issues. If `TypeEq` is to return `HTrue`, the types must be ground; `TypeEq` can return `HFalse` even for unground types, provided they are instantiated enough to determine that they are not equal. So, `TypeEq` is total for ground types, and partial for unground types. We also define the class `TypeCast x y`: a constraint that holds only if the two types `x` and `y` are unifiable. Regarding groundness of `x` and `y`, the class `TypeCast` is less restricted than `TypeEq`. That is, `TypeCast x y` succeeds even for unground types `x` and `y` in case they can be made equal through unification. `TypeEq` and `TypeCast` are related to each other as follows. Whenever `TypeEq` succeeds with `HTrue`, `TypeCast` succeeds as well. Whenever `TypeEq` succeeds with `HFalse`, `TypeCast` fails. But for unground types, when `TypeCast` succeeds, `TypeEq` might fail. So the two complement each other for unground types. Also, `TypeEq` is a partial predicate, while `TypeCast` is a relation. That's why both are useful.

A representation-based equality predicate

The predicate `TypeEq x y b` was introduced in Sec. 6 as follows:

```
class HBool b => TypeEq x y b | x y -> b
```

We now need to provide instances of the class. A very naive implementation would be to explore all combinations of all possible types; see the HLIST source distribution for an illustration. Albeit being portable (Haskell 98 + multi-parameter classes), this leads to an impractical, exponential explosion in the number of instances. A more scalable approach is to introduce a family of infinite types for type-level type representations. That is, we associate types with type representations via a bijection, and we make sure that type representations are more easily compared than the types themselves. We already have all tools for constructing the family of type representations: we can associate with each type constructor an `HNat`, and associate with each type term an `HList` of the representations for the type constructor and its arguments. For instance, using '0' for `Bool`, '1' for `Int`, '2' for `->`, we obtain:

```
class TTypeable a b | a-> b
instance TTypeable Bool (HCons HZero HNil)
instance TTypeable Int  (HCons (HSucc HZero) HNil)
instance (TTypeable a al, TTypeable b bl)
 => TTypeable (a->b) (HCons (HSucc (HSucc HZero))
                            (HCons al (HCons bl HNil)))
```

Because these type representations are constructed in a regular way with ever-increasing naturals, it is sufficient to accommodate type-level equality such that it can compare heterogeneous lists of type-level naturals. Type-level equality for naturals was given in Sec. 4. Here are the remaining instances for `HNil` and `HCons`:

```
instance HEq HNil HNil HTrue
instance HList l => HEq HNil (HCons e l) HFalse
instance HList l => HEq (HCons e l) HNil HFalse
instance ( HList l, HList l'
         , HEq e e' b, HEq l l' b', HAnd b b' b''
         ) => HEq (HCons e l) (HCons e' l') b''
```

All the involved functionality does not go beyond Haskell 98 and multi-parameter classes with uni-directional functional dependencies. GHC and Hugs readily support this combination.

We can now define the class `TypeEq`, using the following instance:

```
instance ( TTypeable t tt, TTypeable t' tt'
         , HEq tt tt' b ) => TypeEq t t' b
```

We make use of a generic instance, which is a common Haskell 98 extension. In turns out that we have essentially transposed what's known as the `Data.Typeable` approach [19] to the type level. We share the drawback of this approach: we need to define an instance of `TTypeable` for each new type constructor. When adding new instances, we have to maintain the bijection between types and type representations. On the other hand, the remaining code is fully generic and does not need to be amended at all.

A generic type equality predicate

We have seen that we can implement `TypeEq` in a portable and even practically usable way, using only commonly supported Haskell extensions. We would like to introduce a fully generic approach, which does not need to be amended when a new type constructor is introduced. Alas, this elegant approach leads us out of the safe haven into uncharted waters of experimental extensions.

The most concise implementation reuses the overlapping tricks that were discussed in Sec. 6, which makes the solution GHC-specific:

```
instance TypeEq x x HTrue
instance (HBool b, TypeCast HFalse b)
     =>  TypeEq x y b
```

Here we take advantage of `TypeCast`, which we define next.

Reification of type unification

The class `TypeCast` was introduced in Sec. 6 and further employed in Sec. 7. `TypeCast x y` differs from just type equality `TypeEq x y HTrue` as follows. If `TypeCast x y` succeeds, then the two types are unified. The difference between unification and just equality emerges when the types are not grounded, i.e., when they contain uninstantiated type variables. The types `[a]` (e.g., of the polymorphic constant `[]`) and `[Bool]` are unifiable, but they are not equal. `TypeEq` cannot establish equality for ungrounded types; however it can establish disequality in case the schemas are sufficiently instantiated to determine that they are not equal.

The most generic implementation of `TypeCast`, which works for both Hugs and GHC, is as follows:

```
instance TypeCast x x where typeCast = id
```

For this implementation to work, we need to import it at a higher level in the module hierarchy than all clients of the class `TypeCast`. Otherwise, type simplification will turn constraints of the form `TypeCast x y` into the form `TypeCast x x`, and thereby inline the unification. We refer to App. D, where we give another implementation of `TypeCast`, which does not require separate compilation. This time, we effectively delay the simplification step with the help of two auxiliary classes. It seems that this delay of type simplification is at the core of all attempts at type-safe cast or type equality (e.g., [4]).

A specific property of our `TypeCast` is that it allows us to control type improvement on a per-instance basis, as the polymorphism benchmark for TIPs showed in Sec. 7. So the utility of `TypeCast` goes strictly beyond a generic implementation of `TypeEq`.

10 Related work

Heterogeneous lists

Type-level list-processing is a relatively obvious opportunity once we get hold on faked dependently typed programming in Haskell, as pioneered by Hallgren and McBride [12, 21]. For instance, homogeneous type-level vectors are considered in [21]. The idea of heterogeneous type-level constructors (what we call `HNil` and `HCons`) occurs elsewhere in the literature. In App. H of [9], Duck et al. motivate their CHR-based model of functional dependencies by operating on such lists using numeral-based access (similar to our's in Sec. 4); Sulzmann also gives a related implementation in the Haskell-style language Chameleon [33]. In [22, 23], Neubauer et al. motivate Haskell extensions for a functional notation of functional dependencies, and for functional logic overloading. The authors consider examples like type-level functions *append* and *length*, as well as record-like operations. By contrast, our goal was to explore the various kinds of access operations for heterogeneous collections: list processing, numeral-based, label-based, and type-based operations. HLIST is the first heterogeneous collection library to the best of our knowledge.

Type-indexed rows

Shield and Meijer have studied the type theory of extensible records and variants starting from a more basic principle, namely type-indexed rows (TIRs) [31]. A TIR is nothing but a type expression that enumerates types. This resembles `HLists`, but TIRs do not comprise any values. So we could go for constructor-less datatypes:

```
class TIR r
data Empty; instance TIR Empty
data e :#: r; instance TIR r => TIR (e :#: r)
```

A TIR is well-formed if the enumerated types are distinct. Well-formedness corresponds to our `HTypeIndexed` constraint. Shield and Meijer provide type-level operators *ALL* and *ONE* that, given a TIR, derive types for type-indexed products (TIPs; recall Sec. 7) and type-indexed co-products (TICs; see App. C for the HLIST implementation of TICs). We could redefine our datatypes for TIPs and TICs such that they take a TIR as parameter, but these definitions and their usage would be more complicated in Haskell. Shield and Meijer argue that, conceptually, a newtype-like mechanism is sufficient for labelling. Our development provides labels as first-class citizens, and we can provide labelled collections without reference to general type-indexing (i.e., numeral indexing is sufficient). Our Haskell-based reconstruction of TIPs and TICs does not require new language extensions.

Extensible records

Foundations of extensible records have been studied intensively. Several Haskell language extensions have been proposed [10, 31, 29], alike for other languages, e.g., (S)ML [6, 30]. There are also record calculi by Bracha, Ohori and others [5, 24]. There are related type systems, e.g., for relational algebra [14]. We have shown that we can reconstruct extensible records in Haskell starting from simpler notions; in particular: heterogeneous lists and equality and of type-level naturals. We cover all typical record operations. We have also defined subtyping constraints in our framework.

Labels, values and records are all first-class citizens in HLIST. So we can write abstractions that take and produce entities of all these kinds. For instance, here is an operation to rename a record label:

```
hRenameLabel l l' r = r'' where
  v   = r  .@. l        -- look up by label
  r'  = r  .-. l        -- delete at label
  r'' = l' .=. v .*. r' -- add new label, old value
```

Type equality and type cast

In our development of heterogeneous collections, we rely on observability of type equality. Also, we employed a reified type unification ('type-level type cast) in a few places. Related expressiveness has been studied in the context of intensional polymorphism [13], dynamic typing [2, 3], and universal representations [36]. Some more recent Haskell-biased work on these notions [34, 8, 4] is not directly usable for our purposes. These approaches either require the programmer to use type representations, or they make a closed-world assumption with regard to the covered types, or they are focused on sums-of-products (as opposed to the immediate coverage of Haskell's newtypes and datatypes), or they involve existential quantification (which makes it difficult to perform more arbitrary operations on elements in the collections). Most notably, we require a type cast that is resolved at type-checking time; run-time would be too late.

Haskell's type classes

Multi-parameter classes [7, 15, 16, 28] with functional dependencies [17, 9] are crucial for type-level programming in Haskell. These typing notions are reasonably understood. There is an ongoing debate if instance selection should be programmable by using constraint-handling rules or functional logic evaluation [32, 23]. Also, the mere notation for encoding type-level functions could perhaps be improved [22]. We have considered using overlapping instances for the definition of some access operations, but ultimately we eliminated use of this debated extension in a systematic manner.

Statically enforced invariants

The `TIP` newtype is an example of a data structure with a statically checked invariant (i.e., uniqueness). Okasaki and others have worked on statically assuring invariants of complex data types, e.g.,

that a matrix is square [25]. These examples normally rely on cleverly chosen data constructors, which make it impossible to construct "wrong" data structures. Our approach is different: type classes let us impose static constraints irrespective of data constructors. Indeed, we use the same data constructor `HCons` to build heterogeneous lists with and without duplicates. We express the constraints in types (sometimes, in phantom types). Our approach does not require extraordinary cleverness in the design of data representation. Furthermore, in the case of constraints encoded in phantom types, there is no run-time or -space overhead of storing and traversing chains of data constructors (`TIP` is just as efficient as `HList`). Because `TIP` is essentially `HList`, we were able to trivially lift all list-processing functions to `TIP`s. Statically checking complex invariants on data structures, such as well-formedness of red-black trees and size-boundaries of lists, is a known application of dependently typed programming [35]. The latter requires non-trivial extensions to a programming language. We have shown that certain invariants, e.g., size boundaries for `HList`s, or uniqueness in `TIP`s, can be statically expressed in Haskell's type system already.

11 Conclusion

We have systematically developed a Haskell library over strongly-typed data structures for heterogeneous collections — lists, arrays, extensible records, and others. The composition of such a data structure, e.g., the types of all elements, is manifest in its type. This makes it possible to strongly type the operations on collections, e.g., look-ups, updates, insertions, and projections. The name of the library, HLIST, emphasises that all data structures are built from typeful heterogeneous lists. We have defined restricted collections, e.g., TIPs, constrained by the requirement that no two elements may have the same type. The constraints are again manifest in the type of the collections and are enforced by the type checker.

The immediate application of our HLIST library is a database access library that covers SQL92, returns the query results as a stream of records, and statically checks that all the queries are consistent with the database schema.

The implications of the library HLIST turn out far reaching, and are still under active investigation. Our TIPs and records are extensible and offer subtyping polymorphism. Our records have first-class labels that can be reused across several record types. We notice that HLIST is implemented in Haskell with only common extensions. Hence the HLIST library addresses the challenge for better Haskell records, without breaking existing programs, as articulated by Simon Peyton Jones at the Haskell Workshop 2003 [11]. Our records also let us implement *has*/*lacks*, record concatenation, length vs. depth subtyping. We can now experiment with these features in real programs — again, without requiring any language extension.

Extensible TIPs and records can be the foundation of the genuine object system. The latter offers subtyping polymorphism (cf. OCaml) as opposed to the class-bounded polymorphism of Haskell. It is remarkable that type classes themselves were instrumental in implementing open TIPs. Extensible records can also be elaborated to provide strongly typed keyword arguments with reusable labels. That is, function arguments can be addressed by keywords, and these arguments can be optional or mandatory. The HLIST source distribution demonstrates keyword arguments. Dual to TIPs are open TICs, offering us dynamics with a statically-checkable constraint on the sort of types encapsulated in the dynamic envelope (cf. App. C). The lists, TIPs, TICs and records of the HLIST library can also be employed in typeful foreign-function interfaces and in XML processing.

Our code relies on the most common Haskell extensions; the use of overlapping instances can be circumvented. In fact, a generic implementation of the predicate `TypeEq` for type equality would still rely on overlapping in a single location. We can also implement `TypeEq` in a portable but non-generic manner relying on one instance per user-defined datatype. Our development suggests that a fundamental solution could be to offer type equality as a primitive in Haskell. We have also identified the utility of reified type unification (or 'type-level type cast') as a tool for type improvement — more fine-grained than functional dependencies. More research is needed to deliver foundational clarifications.

Acknowledgements

We thank Chung-chieh Shan, Martin Sulzmann and the PC of the Haskell Workshop 2004 for very helpful comments and feedback.

12 References

[1] This paper's web site `http://www.cwi.nl/~ralf/HList/`, 2004. This site provides an extended paper version with extra appendicies that could not be included into the Haskell workshop paper. This site also provides a source code distribution for the GHC and Hugs implementations of Haskell.

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *16th ACM Conference on Principles of Programming Languages*, pages 213–227, Jan. 1989.

[3] M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 92–103, San Francisco, June 1992.

[4] A. Baars and S. Swierstra. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

[5] G. Bracha and G. Lindstrom. Modularity Meets Inheritance. In *Proceedings: 4th International Conference on Computer Languages*, pages 282–290. IEEE Computer Society Press, 1992.

[6] F. Burton. Type extension through polymorphism. *TOPLAS*, 12(1):135–138, 1990.

[7] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 170–181. ACM Press, 1992.

[8] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM Press, 2002.

[9] G. Duck, S. Peyton Jones, P. Stuckey, and M. Sulzmann. Sound and Decidable Type Inference for Functional Dependencies. In D. Schmidt, editor, *Proceedings, 13th European Symposium on Programming, ESOP 2004, Barcelona, Spain, March 29 - April 2, 2004*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[10] B. Gaster and M. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical report NOTTCS-TR-96-3, University of Nottingham, Department of Computer Science, Nov. 1996.

[11] H. Nilsson. The Future of Haskell discussion at the Haskell Workshop, 2003. `http://www.mail-archive.com/haskell\@haskell.org/msg13366.html`.

[12] T. Hallgren. Fun with functional dependencies. In *Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Goteborg University, Varberg, Sweden, Jan. 2001*, 2001. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[13] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 130–141. ACM Press, Jan. 1995.

[14] E. W. J. Van den Bussche. Polymorphic type inference for the relational algebra. *Journal of Computer and System Sciences*, 64:694–718, 2002. An extended abstract appeared in PODS'99.

[15] M. Jones. A theory of qualified types. In *Symposium proceedings on 4th European symposium on programming*, pages 287–306. Springer-Verlag, 1992.

[16] M. Jones. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional Programming Languages and Computer Architecture*, pages 160–169. ACM Press, 1995.

[17] M. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000.

[18] O. Kiselyov. Polymorphic stanamically balanced binary trees, 2003. `http://www.haskell.org/pipermail/haskell/2003-April/011621.html`.

[19] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. of the ACM SIGPLAN Workshop TLDI 2003.

[20] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-Specific Languages*, pages 109–122. ACM Press, 1999.

[21] C. McBride. Faking It (Simulating Dependent Types in Haskell), July 2002.

[22] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A Functional Notation for Functional Dependencies. In *Proc. 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy, September 2001*, pages 101–120, 2001.

[23] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. Functional logic overloading. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming languages*, pages 233–244. ACM Press, 2002.

[24] A. Ohori. A polymorphic record calculus and its compilation. *ACM TOPLAS*, 17(6):844–895, 1995.

[25] C. Okasaki. From fast exponentiation to square matrices: an adventure in types. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 28–35. ACM Press, 1999.

[26] C. Okasaki. An Overview of Edison. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier, 2001.

[27] S. Peyton Jones. Adding Ord constraint to instance Monad Set?, 2004. `http://www.haskell.org/pipermail/haskell-cafe/2004-March/005998.html`.

[28] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In J. Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[29] S. Peyton Jones and G. Morrisett. A proposal for records in Haskell, 24 Feb. 2003. Online document: `http://research.microsoft.com/~simonpj/Haskell/records.html`.

[30] D. Remy. Type inference for records in natural extension of ml. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 67–95. MIT Press, 1994.

[31] M. Shields and E. Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 261–275. ACM Press, 2001.

[32] P. Stuckey and M. Sulzmann. A theory of overloading. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 167–178. ACM Press, 2002.

[33] M. Sulzmann et al. Chameleon, 2004. Web site `http://www.comp.nus.edu.sg/~sulzmann/chameleon/`.

[34] S. Weirich. Type-safe cast: (functional pearl). In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 58–67. ACM Press, 2000.

[35] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

[36] Z. Yang. Encoding types in ML-Like languages. In M. Berman and S. Berman, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34, 1 of *ACM SIGPLAN Notices*, pages 289–300, New York, Sept. 27–29 1998. ACM Press.

A Some trivial list-processing operators

We will now transpose several normal list-processing operators to the heterogeneous situation.

Transposition of `head` and `tail`

```
class  HList r => HHead r a | r -> a
 where hHead :: r -> a
instance Fail HListEmpty => HHead HNil ()
 where hHead _ = ()
instance HList r => HHead (HCons a r) a
 where hHead (HCons a _) = a
class (HList r,HList r') => HTail r r' | r -> r'
 where hTail :: r -> r'
instance Fail HListEmpty => HTail HNil HNil
 where hTail _ = HNil
instance HList r => HTail (HCons a r) r
 where hTail (HCons _ r) = r
```

In the above instances, we use the same technique for error messaging as explained in Sec. 6. That is, we employ the `Fail` class to handle invalid applications of the operations. In particular, there is an error message `HListEmpty`, whenever we attempt to access an empty list where a nonempty list is needed. Thus, we have:

```
class Fail x      -- no instances!
data HListEmpty   -- no structure!
```

Transposition of `null`

```
class HBool b => HNull l b | l -> b
instance HNull HNil HTrue
instance HNull (HCons e l) HFalse
```

Transposition of `length`

```
class HNat n => HLength l n | l -> n
instance HLength HNil HZero
instance HLength l n
      => HLength (HCons e l) (HSucc n)
```

B A heterogeneously typed fold operator

We go for the fold operation because it is the ultimate example of a higher-order list-processing function. We dedicate a class `HFoldr` to right-associative folding. The `HFoldr` instances will lift the defining equations for `foldr` to the class level:

```
class HList l => HFoldr f v l r | f v l -> r
 where
  hFoldr :: f -> v -> l -> r
```

The instance for empty lists is trivial:

```
instance HFoldr f v HNil v
 where
  hFoldr _ v _ = v
```

The instance for `HCons` follows the normal `foldr` again, but we assume that function application is modelled by an extra class `HApply`. This allows us to use `hFoldr` for functions that require specific constraints on the involved types:

```
instance (HFoldr f v l r, HApply f (e,r) r')
      =>  HFoldr f v (HCons e l) r'
 where
  hFoldr f v (HCons e l) = hApply f (e,hFoldr f v l)
```

The class `HApply` resembles function application, indeed:

```
class HApply f a r | f a -> r
where
 hApply :: f -> a -> r
```

For instance, we can now redefine `hAppend` in terms of `hFoldr`:

```
hAppend l l' = hFoldr ApplyHCons l' l
```

The datatype `ApplyHCons` stands for "application of `HCons`":

```
data ApplyHCons =
     ApplyHCons -- a proxy for instance selection
```

This meaning of `ApplyHCons` is registered as an `HApply` instance:

```
instance HApply ApplyHCons (e,l) (HCons e l)
 where
  hApply _ (e,l) = HCons e l
```

C Type-indexed co-products

We will now dualise TIPs to arrive at so-called type-indexed sums (or co-products; TICs). A TIC-typed data structure holds a datum of one out of a fixed collection of types. So at the value level, a TIC-typed data structure is not really a collection, but just one datum. However, at the type level we use a list of type proxies to maintain the valid element types of a specific TIC type, and thereby we can restrict construction and destruction of TIC-like data structures.

A TIC demo

We first define an actual TIC type, namely one that models various element types for collections related to the animals in the 'foot-n-mouth' database:

```
type AnimalCol =
  Key :+: Name :+: Breed :+: Price :+: HNil
```

Here we use "`:+:`" rather than "`:*:`" to point out that we are interested in a type-indexed co-product rather than a product. We can now construct actual TIC-like data. For instance:

```
ghci-or-hugs> let myCol = mkTIC Cow :: TIC AnimalCol
```

We can also destruct `myCol`. If we ask for the 'right' type, then destruction succeeds with a result of the form `Just ...`; otherwise we obtain `Nothing`:

```
ghci-or-hugs> unTIC myCol :: Maybe Breed
Just Cow
ghci-or-hugs> unTIC myCol :: Maybe Price
Nothing
```

Most notably, TICs restrict destruction with regard to static typing:

```
ghci-or-hugs> unTIC myCol :: Maybe String
Type error ...
```

Sequences of type proxies

We used the alias "`:+:`" above to enumerate the summands of a TIC type. In fact, "`:+:`" is constructed such that it lines up proxy types in a sequence. Value types would be misleading and confusing here because the sequence of summands is meant for nothing but listing 'options'. So the alias is defined as follows:

```
type e :+: l = HCons (Proxy e) l
```

The actual property of a type sequences to consist only of proxy types is easily specified.

```
class HTypeProxied l
instance HTypeProxied HNil
instance HTypeProxied l
      => HTypeProxied (HCons (Proxy e) l)
```

TICs as constrained dynamics

The demo suggests that a TIC is more constrained than the type `Dynamic`. So in turn, one can define more constrained collection types than just `[Dynamic]` or `String -> Dynamic`. There exist different implementations of TICs, but we will favour here one that indeed directly employs Haskell's dynamics at the value level.

A TIC type is then of the following form:

```
data TIC l = TIC Dynamic -- to be constrained
```

The phantom type parameter `l` of `TIC` enumerates the admitted types that can be injected into this TIC, and that can be subject to extraction attempts. The public constructor for TICs (aka injection) lists all the necessary constraints:

```
mkTIC :: ( HTypeIndexed l
         , HTypeProxied l
         , HOccurs (Proxy i) l
         , Typeable i
         )
       => i -> TIC l
mkTIC i = TIC (toDyn i)
```

The `HTypeIndexed` and `HTypeProxied` constraints require that `l` is a type-indexed sequences of type proxies. The `HOccurs` constraint ensures that the proxy type of the injected value `i` is covered by the sequence of proxies `l`. Finally, the `Typeable` constraint allows us to use Haskell's module `Data.Dynamic`.

It remains to define destruction (or projection), which happens to simply invert the constrained value-to-dynamics conversion:

```
unTIC :: ( HTypeIndexed l
         , HTypeProxied l
         , HOccurs (Proxy o) l
         , Typeable o
         )
       => TIC l -> Maybe o
unTIC (TIC i) = fromDynamic i
```

D Generic type unification cont'd

The class `TypeCast` was described in the subsection 'Reification of type unification' of Sec. 9.

```
class TypeCast a b | a->b, b->a
 where typeCast :: a -> b
```

That section showed the most straightforward implementation of that class: a single instance `TypeCast x x` with the method `typeCast` being just the identity. However, that simple implementation was difficult to use. Separate compilation had to be put to use in some tricky way. Indeed, recall the following example of using `TypeCast` from Sec. 7:

```
instance TypeCast e' e
      => HOccurs e (TIP (HCons e' HNil))
where    hOccurs (TIP (HCons e' _)) = typeCast e'
```

When the compiler sees the instance `TypeCast x x` and combines that with the functional dependencies `a->b`, `b->a` of the class, the compiler infers that the two parameters of `TypeCast` must be the same. That conclusion is correct — the type cast is meant to be an isomorphism on types (in fact, the identity function). What is troublesome is that the type checker applies that conclusion — as a type simplification rule — to the `HOccurs` instance above and infers that `e` must be `e'`. That is a problem however: if a type signature contains distinct type variables, one should be able to instantiate them, at least in principle, with distinct types. Otherwise, the inferred type is less polymorphic than the explicit signature prescribes.

This is the same sort of error that arises in the following code:

```
foo:: a -> b
foo x = x
```

When processing the instance declaration `HOccurs`, the compiler *eagerly* applies the correct type simplification rule – the two parameters of `TypeCast` must be the same – and infers that two type variables `e` and `e'` must be the same. The eagerness creates the problem. We would like to delay the type simplification until after the instance `HOccurs` has been selected and `e` and `e'` have been instantiated. In other words, we would like to unify the *types* that `e` and `e'` are instantiated with, rather than the two *type variables* themselves.

To keep the compiler from applying the type simplification rule too early, we should prevent the early inference of the rule from the instance of `TypeCast` in the first place. For example, we may keep the compiler from seeing the instance `TypeCast x x` until the very end. That is, we place that instance in a separate module and import it at a higher level in the module hierarchy than all clients of the class `TypeCast`. That was the approach described in Sec. 9.

We will now give another implementation of `TypeCast`, which does not require separate compilation. It effectively delays the simplification step with the help of two auxiliary classes.

Our new implementation must keep the semantics of the constraint: `TypeCast a b` should hold if and only if the type corresponding to `a` can be unified with the type corresponding to `b`. On the other hand, we need to allow for polymorphism and pretend that in a constraint `TypeCast a b`, `b` may be something other than `a` — so to keep the typechecker from unifying the type variables `a` in `b` in occurrences of that constraint. Fortunately, the type system is not very smart: when choosing the instances the type-checker looks only at the syntactic form of the type terms involved. Therefore, to fool the type-checker into thinking that `TypeCast a b` is more polymorphic than it really is, we introduce a series of redirections and eventually arrive at the following implementation.

```
class TypeCast'  t a b | t a -> b, t b -> a
 where typeCast'  :: t->a->b
class TypeCast'' t a b | t a -> b, t b -> a
 where typeCast'' :: t->a->b
instance TypeCast'  () a b => TypeCast a b
   where typeCast x = typeCast' () x
instance TypeCast'' t a b => TypeCast' t a b
   where typeCast' = typeCast''
instance TypeCast'' () a a
   where typeCast'' _ x  = x
```

The auxiliary classes `TypeCast'` and `TypeCast''` have an extra, dummy type parameter, which we instantiate to `()` in the instances. Any other ground type would have sufficed. The key to solving the polymorphism quandary is the last instance `TypeCast'' () a a`. It signifies that in the constraint `TypeCast'' t a b`, `b` is *not necessarily* `a`, because `t` can be something other than `()`. Semantically, though, it can never be anything but. However, the type-checker cannot see that and remains satisfied.

Alas, this implementation is specific to GHC; it does not work in Hugs because of the peculiarities of that system with regard to multi-parameter type classes and functional dependencies, which we briefly hinted at in Sec. 6. That shows that multi-parameter type classes with functional dependencies are hard to get right.

While this code works in GHC and is logically sound, we have to admit that we turned the drawbacks of the type-checker to our advantage. This leaves a sour after-taste. We would have preferred to rely on a sound semantic theory of overloading rather than on playing games with the type-checker. Hopefully, the results of the foundational work by Sulzmann and others [32, 23] will eventually be implemented in all Haskell compilers.