Abstract

The library for strongly typed heterogeneous collections HList provides an implementation of extensible records in Haskell that needs only a few common extensions of the language. In HList, records are represented as linked lists of label-value pairs with a lookup operation that is linear-time in the number of fields. In this paper, we use type-level programming techniques to develop a more efficient representation of extensible records for HList. We propose two internal encodings for extensible records that improve lookup at runtime without needing a total order on the labels. One of the encodings performs lookup in constant time but at a cost of linear time insertion. The other one performs lookup in logarithmic time while preserving the fast insertion of simple linked lists. Through staged compilation, the required slow search for a field is moved to compile time in both cases.

Categories and Subject Descriptors
D.3.3 [Programming languages]: Language Constructs and Features; D.1.1 [Programming techniques]: Applicative (Functional) Programming

General Terms
Design, Languages, Performance

Keywords
Extensible Records, Type-level programming, Staged Computation, Haskell, HList, Balanced Trees

1. Introduction

Although there have been many different proposals for Extensible Records in Haskell [5, 9, 10, 14, 15, 19], it is still an open problem to find an implementation that manipulates records with satisfactory efficiency. Imperative dynamic languages use hash tables for objects, achieving constant time insertion and lookup. Inserting a field changes the table in place while preserving the fast insertion of simple linked lists. Through staged compilation, the required slow search for a field is moved to compile time in both cases.

The usual strategies for record insertion in functional languages are copying all existing fields along with the new one to a brand new tuple, or using a linked list [5]. The tuple strategy offers the fastest possible lookup, but insertion is linear time. The linked list sits in opposite in the tradeoff curve, with constant time insertion but linear time lookup. Since a record is essentially a dictionary, the obvious strategy to bridge this gap is a search tree. While lookup is much improved to logarithmic time, insertion is also hit and rendered logarithmic.

Hash maps and ordered trees need hashing and compare functions. This ends up being the biggest turnoff for these techniques in our setting. Types, standing as field labels, do not have natural, readily accessible implementations for these functions.

This paper aims to contribute a solution in that direction. Our starting point is the Haskell library for strongly typed heterogeneous collections HList [13] which provides an example implementation of extensible records. A drawback of HList is that lookup, the most used operation on records, is linear time. We propose two alternative implementations for extensible records as a Haskell library, using the same techniques as HList. One, called ArrayRecord, uses an array to hold the fields, achieving constant time insertions, but lowers lookup to logarithmic time. The other alternative, called SkewRecord, is based on a balanced tree structure. It maintains constant time insertions, but lowers lookup to logarithmic time.

Another contribution of this paper is the trick we use to reduce the run time work. We have observed that, when looking-up an element in a HList, the element is first searched at compile time in order to determine whether it belongs to the list. This search generates the path the program follows at run time to obtain the element. In Figure 1 we represent with a dashed arrow the compile time search, and with a solid arrow the generated path followed at run time. Since the structure is linear, the search and the path have the same length.

Thus, the key idea is very simple. When in Haskell we compare, for example, two stings, such as "foo" ≡ "baar", the entire process of searching the correct instance of Eq to be used is performed at compile time. No work is done at run time to search the correct instance and discard the incorrect ones. We apply the same

---

Copyright notice will appear here once 'preprint' option is removed.
concept to perform the search of a label into a record. Given that a
label is represented by a singleton type we have enough informa-
tion to determine the “path of instances” that goes to it, discarding
any possible wrong path. We also make use of lazy evaluation, to
tell the compiler which path to follow without any cost at run time.

For example, one of our proposed implementations uses an alter-
native structure for the representation of heterogeneous collec-
tions which is based on balanced trees. Such a structure better prof-
ts from the information given by the compile time search, leading
to logarithmic length paths in the run time traversal (see Figure 2).
We show experimental results that confirm this behaviour.

The rest of the paper is organized as follows. We start with a
brief review of the type-level techniques used to implement ex-
tensible records by HList (Section 2). In Section 3 we show how
using the same type-level techniques we can obtain alternative im-
plementations of extensible records with faster lookup operations
at run time. Section 4 presents some experimental results that com-
pare the implementations we propose with HList, both at compile
time and run time. Finally, in Section 5 we draw some conclusions
and present possible directions for future work.

2. HList

HList is a Haskell library that implements typeful heterogeneous
collections, such as heterogeneous lists or records, using extensions
of Haskell for multi-parameter classes [20] and functional depen-
dencies [18]. HList strongly relies on type-level programming tech-
niques by means of which types are used to represent type-level
values, and classes are used to represent type-level functions.

We illustrate the use of type-level programming by means of
two simple examples that will be used later in the paper. We start
with a type-level representation of booleans values. Since we are
only interested in type-level computations, we define empty types
HTrue and HFalse corresponding to each boolean value.

\begin{verbatim}
data HTrue = HTrue
data HFalse = HFalse
\end{verbatim}

The inhabitants hTrue and hFalse of those types are defined
solely to be used in value-level expressions to construct type-level
values by referring to the types of such expressions.

Type-level functions can be described using multi-parameter
classes with functional dependencies. For example, we can encode
type-level negation by defining the following class:

\begin{verbatim}
class HNot t t’ | t -> t’ where
hNot :: t -> t’
\end{verbatim}

The functional dependency \( t \to t’ \) expresses that the parameter \( t \)
uniquely determines the parameter \( t’ \). Therefore, once \( t \) is instan-
tiated, the instance of \( t’ \) must be uniquely inferable by the type-
system. In other words, the relation between \( t \) and \( t’ \) is actually
a function. Whereas the class definition describes the type signa-
ture of the type-level function, the function itself is defined by the
following instance declarations:

\begin{verbatim}
instance HNot HFalse HTrue where hNot _ = hTrue
instance HNot HTrue HFalse where hNot _ = hFalse
\end{verbatim}

If we write the expression \((\text{hNot } h\text{False})\), then we know that \( t \) is
HFalse. So, the first instance of \( \text{hNot} \) is selected and thus \( t’ \) is
inferred to be \( \text{HTrue} \). Observe that the computation is completely at
the type-level; no interesting value-level computation takes place.

Another example is the type-level representation of the maybe
type. In this case we are interested in manipulating a value-level
value associated with each type constructor.

\begin{verbatim}
data HNothing = HNothing
data HJust e = HJust e deriving Show
\end{verbatim}

We aim to construct a type-level value of the maybe type from a
boolean. For this purpose we define the following multi-parameter
class. The parameter \( v \) specifies the type of the values to be con-
tained by a \( \text{HJust} \).

\begin{verbatim}
class HMakeMaybe b v m | b v \to m where
hMakeMaybe :: b \to v \to m
instance HMakeMaybe HFalse v HNothing where
hMakeMaybe b v = HNothing
instance HMakeMaybe HTrue v (HJust v) where
hMakeMaybe b v = HJust v
\end{verbatim}

Another operation that will be of interest on this type is the one
that combines two values of type maybe.

\begin{verbatim}
class HPlus a b c | a b \to c where
hPlus :: a \to b \to c
instance HPlus (HJust v) (HJust v) where
hPlus a c = a
instance HPlus HNothing b c where
hPlus _ _ = b
\end{verbatim}

2.1 Heterogeneous Lists

Heterogeneous lists can be represented with the data types \(HNil\)
and \(HCons\), which model the structure of lists both at the value
and type level:

\begin{verbatim}
data HNil = HNil
data HCons e l = HCons e l
infixr 2 ': '
\end{verbatim}

For example, the value \(\text{HCons True} \ (\text{HCons } 'a' \ \text{HNil})\) is a
heterogeneous list of type \(\text{HCons Bool} \ (\text{HCons Char } \text{HNil})\).

2.2 Extensible Records

Records are mappings from labels to values. They are modeled by
an \(\text{HList}\) containing a heterogeneous list of fields. A field with
label \( l \) and value of type \( v \) is represented by the type:

\begin{verbatim}
newtype Field l v = Field \{ value :: v \}
\end{verbatim}

Notice that the label is a phantom type [7]. We can retrieve the label
value by using the function \(\text{label}\), which exposes the phantom type
parameter:

\begin{verbatim}
label :: Field l v \to l
label = \_
\end{verbatim}

We define separate types and constructors for labels.

\begin{verbatim}
data L1 = L1
data L2 = L2
data L3 = L3
data L4 = L4
\end{verbatim}
data L5 = L5
data L6 = L6
data L7 = L7

Thus, the following defines a record (rList) with seven fields:

\[
\begin{align*}
    rList = &\quad (L1 \approx \text{True}) \cdot \text{HCons}' \\
    &\quad (L2 \approx 9) \cdot \text{HCons}' \\
    &\quad (L3 \approx \text{"bla"}) \cdot \text{HCons}' \\
    &\quad (L4 \approx \text{"t"}) \cdot \text{HCons}' \\
    &\quad (L5 \approx \text{Nothing}) \cdot \text{HCons}' \\
    &\quad (L6 \approx [4, 5]) \cdot \text{HCons}' \\
    &\quad (L7 \approx \text{"last"}) \cdot \text{HCons}' \\
    &\quad \text{HNil}
\end{align*}
\]

The class \textit{HListGet} retrieves from a record the value part corresponding to a specific label:

\texttt{class HListGet } r \ x \ v \mid r \ l \rightarrow v \ \texttt{where}
\texttt{hListGet :: } r \rightarrow l \rightarrow v

At the type-level it is statically checked that the record \( r \) indeed has a field with label \( l \) associated with a value of the type \( v \). At value-level \texttt{HListGet} returns the value of type \( v \). For example, the following expression returns the string "last":

\[
\text{lastList} = \text{hListGet } rList \ L7
\]

Instead of polluting the definitions of type-level functions with the overlapping instance extension when comparing two types to be equal (e.g. labels), \texttt{HList} encapsulates type comparison in \texttt{HEq}. The type equality predicate \texttt{HEq} results in \texttt{HTrue} in case the compared types are equal and \texttt{HFalse} otherwise. Thus, when comparing two types in other type-level functions (like \texttt{HListGet} below), these two cases can be discriminated without using overlapping instances.

\[
\text{class } \texttt{HEq } x \ y \ b \mid x \rightarrow y \rightarrow b
\]

\[
\begin{align*}
\text{hEq} &\approx \text{HEq } x \ y \ b \Rightarrow x \rightarrow y \rightarrow b \\
\text{hEq} &\approx \bot
\end{align*}
\]

We will not delve into the different possible definitions for \texttt{HEq}. For completeness, here is one that suffices for our purposes. For a more complete discussion about type equality in Haskell we refer to [11].

\[
\begin{align*}
\text{instance } \texttt{HEq } x \ x \ HTrue & \\
\text{instance } b \sim \texttt{HFalse} &\Rightarrow \texttt{HEq } x \ y \ b
\end{align*}
\]

At this point we can see that the use of overlapping instances is unavoidable. This explains why the implementation of \texttt{HList} is based on type classes and functional dependencies instead of type families [3, 4, 21] (which do not support overlapping instances).

\texttt{HListGet} uses \texttt{HEq} to discriminate the two possible cases. Either the label of the current field matches \( l \), or the search must continue to the next node.

\[
\begin{align*}
\text{instance } &\approx \texttt{HEq } l \ t \ b \\
&\quad \texttt{HListGet'} b \ v' \ r' \ l \ v \Rightarrow \\
&\quad \texttt{HListGet } (\texttt{HCons } \texttt{(Field } l' \ v') \ r') \ l \ v \ \texttt{where} \\
&\quad \texttt{hListGet } (\texttt{HCons } f'@\texttt{(Field } v') \ r') \ l \ = \\
&\quad \texttt{HListGet'} (\texttt{hEq } l \ (\texttt{label } f')) \ v' \ r' \ l
\end{align*}
\]

\texttt{HListGet'} has two instances, for the cases \texttt{HTrue} and \texttt{HFalse}.

\[
\begin{align*}
\text{class } \texttt{HListGet'} b \ v' \ r' \ l \ v \mid b \ v' \ r' \ l \rightarrow v \ \texttt{where} \\
&\quad \texttt{hListGet'} :: b \rightarrow v' \rightarrow r' \rightarrow l \rightarrow v
\end{align*}
\]

\[
\begin{align*}
\text{instance } &\approx \texttt{HListGet'} \ HTrue \ v \ r' \ l \ v
\end{align*}
\]

where

\[
\begin{align*}
\texttt{HListGet'} \ _ \ v \ _ \ = \ v
\end{align*}
\]

\texttt{instance}

\[
\begin{align*}
\texttt{HListGet'} r' \ l \ v \Rightarrow \\
\texttt{HListGet'} \ \texttt{HFalse } v' \ r' \ l \ v \ \texttt{where}
\end{align*}
\]

\[
\begin{align*}
\texttt{hListGet'} \ _ \ r' \ l = \texttt{hListGet'} \ r' \ l
\end{align*}
\]

If the labels match, the corresponding value is returned, both at the value and type levels. Otherwise, \texttt{HListGet'} calls back to \texttt{HListGet} to continue the search. The two type-functions are mutually recursive. There is no case for the empty list; lookup fails.

For GHC, the type level machinery not only generates correct value level code, but efficient code too. At the value level, the functions \texttt{hListGet} and \texttt{hListGet'} are trivial, devoid of logic and conditions. For this reason, GHC is smart enough to elide the dictionary objects and indirect jumps for \texttt{hListGet}. The code is inlined to a case cascade, but the program must traverse the linked list. For example, this is the GHC core of the example:

\[
\begin{align*}
\texttt{lastListCore} &\approx \texttt{case } rList \texttt{ of } \\
&\quad \texttt{HCons } \texttt{rs1 } \rightarrow \texttt{case } \texttt{rs1} \texttt{ of } \\
&\quad \texttt{HCons } \texttt{rs2 } \rightarrow \texttt{case } \texttt{rs2} \texttt{ of } \\
&\quad \texttt{HCons } \texttt{rs4 } \rightarrow \texttt{case } \texttt{rs4} \texttt{ of } \\
&\quad \texttt{HCons } \texttt{rs5 } \rightarrow \texttt{case } \texttt{rs5} \texttt{ of } \\
&\quad \texttt{HCons } \texttt{rs6 } \rightarrow \texttt{case } \texttt{rs6} \texttt{ of } \\
&\quad \texttt{HCons } \rightarrow \texttt{v}
\end{align*}
\]

3. Faster Extensible Records

Extensible records can double as “static type-safe” dictionaries, that is, collections that guarantee at compile time that all labels searched for are available. For example, [24], a library for first-class attribute grammars, uses extensible records to encode the collection of attributes associated to each non-terminal. If we wanted to use it to implement a system with a big number of attributes (e.g. a compiler) an efficient structure would be needed. Increasing the size of GHC’s context reduction stack makes the program compile but at run time the linear time lookup algorithm hurts performance. The usual replacement when lookup in a linked list is slow is a search tree. In that case we would need to define a \texttt{HOrd} type-function analogue to \texttt{HList}’s \texttt{magc, HEq} and port some standard balanced tree to compile time, tricky rotations and all. As unappealing as this already is, the real roadblock is \texttt{HOrd}. Without help from the compiler, defining such type function for unstructured labels is beyond (our) reach.

The key insight is that sub-linear behavior is only needed at run time. We do not worry if the work done at compile time is superlinear as long as it helps us to speed up our programs at run time. \texttt{HListGet} already looks for our label at compile time to fail compilation if we require a field for a record without such label. So our idea is to maintain the fields stored unordered, but in a structure that allows fast random access and depends on the compiler to hardcode the path to our fields.

We will present two variants of faster records. To make code listing shorter and easier to understand, we implement each variant with independent interfaces. However, it would be possible to provide a common class-based interface for all variants.

The first variant follows the conventional approach of storing the record as a tuple. However, because Haskell does not offer generivity over the length of tuples as in [23], i.e. efficient access to the \( i \)-th element of an arbitrary length tuple, we will use an array instead, converting field values to a common type. This implementation supports linear time insertions and constant time lookups.
The second variant is tree-like, being based on Skew Binary Random-Access Lists [17], a structure that guarantees constant time insertions and logarithmic time access to any element. Other, perhaps simpler, data structures such as Braun trees [8] could have been chosen, since the key property of searching at compile time while retrieving at run time works unchanged in any balanced tree structure. However, those structures do not offer constant time insertion and are not drop-in replacements for simple linear lists. A structure with logarithmic insertion slows down applications heavy on record modification.

### 3.1 Array Records

An Array Record has two components: an array containing the values of the fields, and an heterogeneous list used to find a field’s ordinal for lookup in the array. To allow the storage of elements of different types in the array, we use the type Any. Items are then unsafeCoerce on the way in and out based on the type information we keep in the heterogeneous list.

```haskell
data ArrayRecord r = ArrayRecord r (Array Int Any)
```

#### 3.1.1 Lookup

Lookup is done as a two step operation. First, the ordinal of a certain label in the record, and the type (\(o\)) of its stored element, are found with `ArrayFind`.

```haskell
class ArrayFind r l v | r l \rightarrow v where
  arrayFind : r \rightarrow l \rightarrow Int
```

Second, function `hArrayGet` uses the index to obtain the element from the array and the type (\(v\)) to coerce that element to its correct type.

```haskell
hArrayGet :: ArrayFind r l v \implies ArrayRecord r \rightarrow l \rightarrow v
hArrayGet (ArrayRecord r a) l = unsafeCoerce (a ! arrayFind r l)
```

Figure 3 shows a graphical representation of this process. Dashed arrow represents the compile time search of the field in the heterogeneous list which results in the index of the element in the array. Using this index the element is retrieved from the array in constant time at run time (solid arrow).

`ArrayFind` follows the same pattern as `HListGet` shown earlier, using `HEq` to discriminate the cases of the label of the current field, which may match or not the searched one.

```haskell
instance (HEq l l’ b
  , ArrayFind’ b v’ l v n
  , ToValue n) \implies
  ArrayFind (HCons (Field l’ v’) r) l v where
```

A difference with `HListGet` is that the work of searching the label, performed by `ArrayFind'`, is only done at type-level. There is no value-level member of the class `ArrayFind'`; observe that `ArrayFind'` is just an undefined value and nothing will be computed at run time.

```haskell
arrayFind (HCons f r) l =
  toValue (arrayFind' (HEq l (label f)) (value f) r l)
```

To perform this conversion in constant time, we have to provide one specific instance of `ToValue` for every type-level natural we use.

```haskell
instance ToValue n where
toValue :: n \rightarrow Int
```

In this implementation of `ArrayFind` it is very easy to distinguish the two phases of the lookup process. However, the use of the function `toValue` introduces a big amount of boilerplate. Although these instances can be automatically generated using Template Haskell, we make use of a couple of optimizations that are present in GHC to propose a less verbose implementation of `ToValue`.

```haskell
instance ToValue HZero where
  toValue _ = 0
instance ToValue (HSucc HZero) where
  toValue _ = 1
instance ToValue (HSucc (HSucc HZero)) where
  toValue _ = 2
...
```

Based on inlining and constant folding, the computation of the index, which is linear time, is performed at compile time.

#### 3.1.2 Construction

An empty `ArrayRecord` consists of an empty heterogeneous list and an empty array.

```haskell
emptyArrayRecord = ArrayRecord HNil (array 0 (−1) [])
```

Function `hArrayExtend` adds a field to an array record.

```haskell
hArrayExtend f = hArrayModifyList (HCons f)
```
hArrayModifyList hc (ArrayRecord r _) =
  let r' = hc r
  fs = hMapAny r'
  in ArrayRecord r' (listArray (0, length fs - 1) fs)

The new field (which includes the type information of the element) is added to the heterogeneous list of the old record. The extended heterogeneous list is then converted to a plain Haskell list with hMapAny and turned into the array of the new record with listArray. Note that the array of the old record is not used. In this way, if several fields are added to a record but lookup is not done on the intermediate records, the intermediate arrays are not ever created by virtue of Haskell’s laziness. Adding n fields is then a linear time operation instead of quadratic. This optimization is the reason why an ArrayRecord contains the actual corresponding HList instead of just the field value type relation as a phantom parameter (i.e. only at the type-level). The function hMapAny iterates over the heterogeneous list coercing its elements to values of type Any.

class HMapAny r where
  hMapAny :: r -> [Any]
instance HMapAny HNil where
  hMapAny _ = []
instance
  HMapAny r ⇒
  HMapAny (HCons (Field l v) r)
  where
  hMapAny (HCons (Field v) r) = unsafeCoerce v : hMapAny r

3.1.3 Update and Remove

Functions hArrayUpdate and hArrayRemove, to update and remove a field respectively, are similar to the extension function in the sense that both have to reconstruct the array after modifying the list. We use the respective functions hListUpdate and hListRemove from the HList implementation of records.

hArrayUpdate l e
  = hArrayModifyList (hListUpdate l e)

hArrayRemove l
  = hArrayModifyList (hListRemove l)

With HArrayUpdate we change a field of some label with a new field with possibly new label and value.

3.2 Skew Binary Random-Access List

We start with a description of Skew Binary Random-Access List [16] in a less principled but easier and more direct fashion than [17], which is founded on numerical representations. A skew list is a linked list spine of complete binary trees.

The invariant of skew lists is that the height of the trees get strictly larger along the linked list, except that the first two trees may be of equal size. Because of the size restriction, the spine is bounded by the logarithm of the element count, as is each tree. Hence, we can get to any element in logarithmic effort. This is a fundamental property of skew lists we will take advantage of.

Insertion maintaining the invariant is constant time and considers two cases: (1) when the spine has at least two trees and the first two trees are of equal size, we remove them and insert a new node built of the new element and the two trees removed; and (2) we just insert a new leaf. In Figure 4 we show a graphic representation of the successive skew lists that arise in the process of construction of a skew list with the elements of rList from section 2.2. Nodes connected by arrows represent linked-lists and nodes connected by lines represent trees. The first two steps (adding elements with label l7 and l6) are in case (2), thus two leaves are inserted into the spine. On the other hand, the third step (adding an element with label l5) is in case (1), so a node has to be built with the new element as root and the two previous trees as subtrees.

Skew lists are not optimal for merging records. In the view of tree instances as numbers, merging is equivalent to number addition. Some priority queue structures do support fast merging (or melding), but usually the resulting trees are very deep and do not support efficient access to some elements.

3.3 SkewRecord

In this subsection we present our implementation of extensible records using (heterogeneous) skew lists. First, we introduce some types to model heterogeneous binary trees:

data HEmpty = HEmpty

data HNode e t t' = HNode e t t'

type HLeaf e = HNode e HEmpty HEmpty

and a smart constructor for leaves:

hLeaf e = HNode e HEmpty HEmpty

The element precedes the subtrees in HNode so all elements in expressions read in order left to right. The common leaf case warrants the helper type HLeaf and the smart constructor hLeaf.

A (heterogeneous) skew list is then defined as a heterogeneous list of (heterogeneous) binary trees. The following declarations define a skew list with the elements of the fourth step of Figure 4:

four =
  HCons (hLeaf (L4 := [c']))  §
  HCons (HNode (L5 := Nothing)
    (hLeaf (L6 := [4, 5]))
    (hLeaf (L7 := "last")))  §
  HNil

3.3.1 Construction

We define a smart constructor emptySkewRecord for empty skew lists, i.e. an empty list of trees.

emptySkewRecord = HNil

HHeight returns the height of a tree. We will use it to detect the case of two leading equal height trees in the spine.

class HHeight t h | t → h
instance HHeight HEmpty HZero

Figure 4. Insertion in a Skew
instance HHeight t h ⇒
HHeight (HNode e t t') (HSuc h)

HskewCarry finds out if a skew list l is in case (1) or (2). This will be used for insertion to decide whether we need to take the two leading existing trees and put them below a new HNode (case 1), or just insert a new HLeaf (case 2). In the numerical representation of data structures, adding an item is incrementing the number. If each top level tree is a digit, building a new taller tree is a form of carry, so HskewCarry returns HTrue.

class HskewCarry l b | : → b
hskewCarry :: HskewCarry l b ⇒ l → b
hskewCarry = ↓

If the spine has none or one single tree we return HFalse.

instance HskewCarry HNil HFalse
instance HskewCarry (HCons t HNil) HFalse

In case the spine has more than one tree, we return HTrue if the first two trees are of equal size and HFalse otherwise.

instance (HHeight t h , HHeight t' h' , HEq h h' b) ⇒
HskewCarry (HCons t (HCons t' bs)) b

All these pieces allow us to define HskewExtend, which resembles the HCons constructor.

class HskewExtend f r r' | f r → r'
where hskewExtend :: f r → r' → r'

infixr 2 'hskewExtend'

HskewExtend looks like HListGet shown earlier. HskewCarry is now responsible for discriminating the current case, while HListGet used HEq on the two labels.

instance (HskewCarry r b , HskewExtend' b f r r') ⇒
HskewExtend f r r' where
hskewExtend f r =

hskewExtend' (hskewCarry r) f r

class HskewExtend' b f r r' | b f r → r' where
hskewExtend' :: b → f r → r'

Here HFalse means that we should not add up the first two trees of the spine. Either the size of the two leading trees are different, or the spine is empty or a singleton. We just use HLeaf to insert a new tree at the beginning of the spine.

instance HskewExtend'
HFalse
f
r

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.

instance HskewExtend'
HTrue

f

(HCons (HLeaf f) r) where
hskewExtend' – f r = HCons (hLeaf f) r


hskewExtend' – f r = HCons (hLeaf f) r

When HskewCarry returns HTrue, however, we build a new tree reusing the two trees that were at the start of the spine. The length of the spine is reduced in one, since we take two elements but only add one.
comparing both labels. We call $H\text{MakeMaybe}$ with the result of the comparison, and $H\text{Nothing}$ or $H\text{Just}$ is returned as appropriate.

$$\text{instance (HEq l l' b, H\text{MakeMaybe} b v m) ⇒ H\text{SkewGet} (Field l' v) l m \ where h\text{SkewGet} f l = \ h\text{MakeMaybe (HEq l (label f))} (value f)}$$

When we repeat the experiment at the end of subsection 2.2, but constructing a $\text{SkewRecord}$ instead of an $\text{HList}$:

$$r\text{Skew} =\begin{cases} \text{(L1 := True) 'hSkew\text{Extend'}} & \\
\text{(L2 := 9) 'hSkew\text{Extend'}} & \\
\text{(L3 := "bla") 'hSkew\text{Extend'}} & \\
\text{(L4 := 'c') 'hSkew\text{Extend'}} & \\
\text{(L5 := Nothing) 'hSkew\text{Extend'}} & \\
\text{(L6 := [4, 5]) 'hSkew\text{Extend'}} & \\
\text{(L7 := "last") 'hSkew\text{Extend'}} & \\
\text{empty\text{SkewRecord}} & \\
\end{cases}$$

$$\text{last\text{Skew} = case r\text{Skew} of HCons t1 l f ⇒ case t1 of HNode _ _ t12 ⇒ case t12 of HNode _ _ t121 ⇒ case t121 of HNode e _ _ ⇒ e}$$

Thus, getting to $l7$ at run time only traverses a (logarithmic length) fraction of the elements, as we have seen in Figure 2. Later we will examine runtime benchmarks.

### 3.3.3 Update

We now define an update operation that makes it possible to change a field of some label with a new field with possibly new label and value.

$$\text{class H\text{SkewUpdate} l e r r' | l e r → r' where h\text{SkewUpdate} :: l e r → r → r'}$$

We use the lookup operation $H\text{SkewGet}$ to discriminate at type-level whether the field with the searched label is present or not in the skew list.

$$\text{instance (H\text{SkewGet} r l m, H\text{SkewUpdate'} m l e r r') ⇒ H\text{SkewUpdate} l e r r' \ where h\text{SkewUpdate} l e r = (h\text{SkewUpdate'} (h\text{SkewGet} r l) l e r)}$$

$$\text{class H\text{SkewUpdate'} m l e r r' | m l e r → r' where h\text{SkewUpdate'} :: m l e r → r → r'}$$

In case the label is not present we have nothing to do than just returning the structure unchanged.

$$\text{instance H\text{SkewUpdate'} H\text{Nothing} l e r r where h\text{SkewUpdate'} _ _ l e r = r}$$

In the other cases (i.e. when lookup results in $H\text{Just}$ $v$) we call $h\text{SkewUpdate}$ recursively on all subparts in order to apply the update when necessary. Because of the previous instance (when lookup returns $H\text{Nothing}$), at run time recursion will not enter in those cases where the label is not present. We start the process in the spine.

$$\text{instance (H\text{SkewUpdate} l e t t', H\text{SkewUpdate} l e t t) ⇒ H\text{SkewUpdate'} (H\text{Just} v) l e (H\text{Cons} t t') \ (H\text{Cons} t t')}$$

where $h\text{SkewUpdate'} _ _ l e (H\text{Cons} t t) =$ $H\text{Cons} (h\text{SkewUpdate} l e t)$ $(h\text{SkewUpdate} l e t)$

On a $H\text{Node}$, $h\text{SkewUpdate}$ is recursively called on the left and right sub-trees as well as on the element of the node.

$$\text{instance (H\text{SkewUpdate} l e e e'' \ H\text{SkewUpdate} l e t t' \ H\text{SkewUpdate} l e t t \ ⇒ H\text{SkewUpdate'} (H\text{Just} v) l e (H\text{Node} e' t t') \ (H\text{Node} e'' t t')}$$

where $h\text{SkewUpdate'} _ _ l e (H\text{Node} e' t t) =$ $H\text{Node} (h\text{SkewUpdate} l e e') (h\text{SkewUpdate} l e t)$ $(h\text{SkewUpdate} l e t)$

Finally, when we arrive to a $H\text{Field}$ and we know the label is the one we are searching for (because we are considering the case $H\text{Just} v$), we simply return the updated field.

$$\text{instance H\text{SkewUpdate'} (H\text{Just} v) l e (H\text{Field} l v) e \ where h\text{SkewUpdate'} _ _ l e e' = e}$$

At run time, this implementation of $h\text{SkewUpdate}$ only rebuilds the path to the field to update, keeping all other sub-trees intact. Thus the operation runs in time logarithmic in the size of the record.

### 3.3.4 Remove

Removing a field is easy based on updating. We overwrite the field we want to eliminate with the first field in the skew list, and then we remove the first field from the list. Thus, we remove elements in logarithmic time while keeping the tree balanced.

First, we need a helper to remove the first element of a skew list.

$$\text{class H\text{SkewTail} ts ts' | ts → ts' where h\text{SkewTail} :: ts → ts'}$$

In Figure 5 we show an example of the possible cases we can find.
The easy case is when the spine begins with a leaf. We just return the tail of the spine list.

\[
\text{instance } \text{HListTail} \ (\text{HCons} \ (\text{HLeaf} \ e) \ ts) \ \text{ts where } \\
\quad \text{HListTail} \ (\text{HCons} \ _ \ ts) = ts
\]

The other case is when the spine begins with a tree of three or more elements. Since \text{HLeaf} is a synonym of \text{HNode} with \text{HEmpty} as sub-trees, we need to assert the case when the sub-trees of the root \text{HNode} are nonempty (i.e. \text{HNodes} themselves). By construction, both sub-trees have the same shape, but doing pattern matching on the first one only suffices to make sure this case does not overlap with the previous one. In this case we grow the spine with the sub-trees, throwing away the root.

\[
\text{instance} \\
\quad \text{HSkewTail} \\
\quad (\text{HCons} \ (\text{HNode} \ e \ t \ (\text{HNode} \ e' \ t') \ ts)) \ (\text{HCons} \ t \ ((\text{HCons} \ (\text{HNode} \ e \ t \ t') \ ts)) \\
\text{where} \\
\quad \text{HSkewTail} \ (\text{HCons} \ (\text{HNode} \ _ \ t') \ ts) = \text{HCons} \ t \ (\text{HCons} \ t' \ ts)
\]

Last, \text{hSkewRemove} takes the first node and calls \text{hSkewUpdate} to duplicate it where the label we want gone was. Then \text{hSkewTail} removes the original occurrence, at the start of the list.

\[
\text{hSkewRemove} \ l \ (\text{HCons} \ (\text{HNode} \ e \ t \ t') \ ts) = \text{hSkewTail} \ \\
\text{hSkewUpdate} \ l \ e \ (\text{HCons} \ (\text{HNode} \ e \ t \ t') \ ts)
\]

4. Efficiency

In order to chose the best implementation in practice and as a sanity check, we did some synthetic benchmarks of the code. We compile and run the programs in a 4 core 2.2 Ghz second generation (Sandy Bridge) Intel i7 MacBook Pro Notebook with 8 GB of RAM. We use GHC version 7.6.1 64 bits under OS X 10.8 Mountain Lion.

We time accessing the last of an increasing number of fields. The program constructs the list once and runs a 10 million iteration lookup loop, taking the necessary precautions to avoid the compiler exploiting the language lazyness to optimize out all our code. Run time comparisons are shown in Figure 6.

Note how in practice \text{ArrayRecord} and \text{SkewRecord} take the same time no matter the length of the record. Actually, sometimes larger records run faster than smaller records for \text{SkewRecord}. For example, a 31 size skew list contains a single tree, so elements are at most 5 hops away. But a 28 size skew lists contains trees sized 1, 1, 3, 7 and 15, and getting to the last takes 8 hops.

Up to ten elements, simple linked lists are faster than skew lists. By fusing the spine list and the tree nodes, skew lists can be tweaked to improve the performance with few elements. This results in a single node type, with an element and three child node references, one to the next node, one to the right subtree, and one to the node of the next tree. We chose the unfused exposition for clarity. Another option is to use linked list for small records and switch to skew list when over 10 fields. Since the test is done at compile time, the adaptive structure has no run time overhead above having to copy the 10 fields from the linked list to the tree when the limit is surpassed.

Next, Figure 7 shows the runtime of inserting one more field to a record of a given length. To force the worst case for \text{ArrayRecord}, we disable the insertion optimization by immediately looking up the field just inserted. The insert-lookup process is run one million times. Only \text{ArrayRecord} is graphed because the other alternatives are too fast in this case. The graph exposes the linear time behavior of \text{ArrayRecord}, its Achilles’ heel. However, we do not expect real life applications to fall in this case. In general, multiple adjacent insertions preceding a lookup would be the common case.

For Figure 8 we compared updating the first and deepest element in each implementation. As expected, \text{SkewRecord} is negligible. \text{HListRecord} is a linear graph picking up somewhat proba-
To improve performance, the code can be rewritten with type families. The main reason why we based our development on functional dependencies is the lack of overlapping instances at type families. In case further investigation on type families solves this problem we would be able to rephrase our implementation in terms of type families with a trivial translation, achieving a more functional style implementation.

An interesting aspect of the proposed approach to extensible records is that it can be encoded as a Haskell library, using only nowadays established extensions implemented for example in current versions of GHC. However, better performance could be achieved if our approach is developed as a built-in implementation in a compiler. In that case, the `ArrayRecord` solution reduces to the standard tuple-based techniques [5]. On the other hand, `SkewRecord` provides a novel encoding with fast lookup and insertion that would preserve its advantages even as a built-in solution.

### References


