Just Do It While Compiling!

Fast Extensible Records In Haskell

Bruno Martinez Aguerre
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
brunom@fing.edu.uy

Marcos Viera
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy

Alberto Pardo
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
pardo@fing.edu.uy

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, destructing the old version of the object, not allowing for persistency as required in functional languages. Copying the underlying array of the hash table to preserve the old version makes insertion slower.

Clojure [6] implements vectors with trees of small contiguous arrays, so insertion is logarithmic due to structural sharing. Clojure's hash map, built on top of vectors, then achieves logarithmic time insertion and lookup.

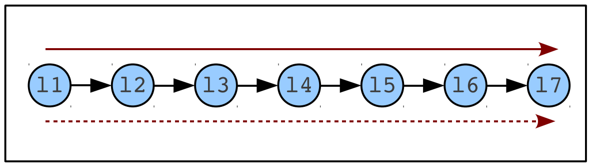

Figure 1. Search $l7$ in HList

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 lookup but linear time insertion. 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"` $\equiv$ `"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

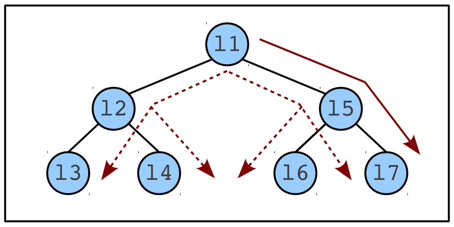

Figure 2. Search $l7$ in balanced tree

concept to perform the search of a label into a record. Given that a label is represented by a singleton type we have enough information 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 alternative structure for the representation of heterogeneous collections which is based on balanced trees. Such a structure better profits 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 extensible records by HList (Section 2). In Section 3 we show how using the same type-level techniques we can obtain alternative implementations of extensible records with faster lookup operations at run time. Section 4 presents some experimental results that compare 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 dependencies [18]. HList strongly relies on *type-level programming* techniques 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.

```
data HTrue ; hTrue  = ⊥ :: HTrue
data HFalse; hFalse = ⊥ :: HFalse
```

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:

```
class HNot t t' | t → t' where
  hNot :: t → t'
```

The functional dependency $t \to t'$ expresses that the parameter t uniquely determines the parameter t'. Therefore, once t is instantiated, 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 signature of the type-level function, the function itself is defined by the following instance declarations:

```
instance HNot HFalse HTrue  where hNot _ = hTrue
instance HNot HTrue  HFalse where hNot _ = hFalse
```

If we write the expression $(hNot\ hFalse)$, then we know that t is $HFalse$. So, the first instance of $hNot$ is selected and thus t' is inferred to be $HTrue$. 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.

```
data HNothing = HNothing
data HJust e   = HJust e deriving Show
```

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 contained by a $HJust$.

```
class HMakeMaybe b v m | b v → m where
  hMakeMaybe :: b → v → m
instance HMakeMaybe HFalse v HNothing where
  hMakeMaybe b v = HNothing
instance HMakeMaybe HTrue v (HJust v) where
  hMakeMaybe b v = HJust v
```

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

```
class HPlus a b c | a b → c where
  hPlus :: a → b → c
instance HPlus (HJust a) b (HJust a) where
  hPlus a _ = a
instance HPlus HNothing b b where
  hPlus _ b = b
```

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:

```
data HNil       = HNil
data HCons e l = HCons e l
infixr 2 ‘HCons‘
```

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

2.2 Extensible Records

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

```
newtype Field l v = Field {value :: v}
(.=.)              :: l → v → Field l v
_ .=. v            = Field v
```

Notice that the label is a phantom type [7]. We can retrieve the label value by using the function $label$, which exposes the phantom type parameter:

```
label :: Field l v → l
label = ⊥
```

We define separate types and constructors for labels.

```
data L1 = L1
data L2 = L2
data L3 = L3
data L4 = L4
data L5 = L5
data L6 = L6
data L7 = L7
```

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

```
rList =
  (L1 .=. True    ) `HCons`
  (L2 .=. 9       ) `HCons`
  (L3 .=. "bla"   ) `HCons`
  (L4 .=. 'c'     ) `HCons`
  (L5 .=. Nothing ) `HCons`
  (L6 .=. [4,5]   ) `HCons`
  (L7 .=. "last"  ) `HCons`
  HNil
```

The class *HListGet* retrieves from a record the value part corresponding to a specific label:

```
class HListGet r l v | r l → v where
  hListGet :: r → l → 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 *hListGet* returns the value of type v. For example, the following expression returns the string `"last"`:

```
lastList = 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), HList encapsulates type comparison in *HEq*. The type equality predicate *HEq* results in *HTrue* in case the compared types are equal and *HFalse* otherwise. Thus, when comparing two types in other type-level functions (like *HListGet* below), these two cases can be discriminated without using overlapping instances.

```
class HEq x y b | x y → b
hEq :: HEq x y b ⇒ x → y → b
hEq = ⊥
```

We will not delve into the different possible definitions for *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].

```
instance                    HEq x x HTrue
instance b ∼ HFalse ⇒ HEq x y b
```

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

HListGet uses *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.

```
instance
  (HEq l l' b
  , HListGet' b v' r' l v) ⇒
   HListGet (HCons (Field l' v') r') l v where
  hListGet (HCons f'@(Field v') r') l =
    hListGet' (hEq l (label f')) v' r' l
```

HListGet' has two instances, for the cases *HTrue* and *HFalse*.

```
class HListGet' b v' r' l v | b v' r' l → v where
  hListGet' :: b → v' → r' → l → v
instance
  HListGet' HTrue v r' l v
  where
  hListGet' _ v _ _ = v
instance
  HListGet r' l v ⇒
  HListGet' HFalse v' r' l v where
  hListGet' _ _ r' l = hListGet r' l
```

If the labels match, the corresponding value is returned, both at the value and type levels. Otherwise, *HListGet'* calls back to *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 *hListGet* and *hListGet'* are trivial, devoid of logic and conditions. For this reason, GHC is smart enough to elide the dictionary objects and indirect jumps for *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:

```
lastListCore = case rList of
  HCons _ rs1 → case rs1 of
    HCons _ rs2 → case rs2 of
      HCons _ rs3 → case rs3 of
        HCons _ rs4 → case rs4 of
          HCons _ rs5 → case rs5 of
            HCons _ rs6 → case rs6 of
              HCons e _ → e
```

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 *HOrd* type-function analogue to HList's magic *HEq* and port some standard balanced tree to compile time, tricky rotations and all. As unappealing as this already is, the real roadblock is *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. *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 genericity 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.

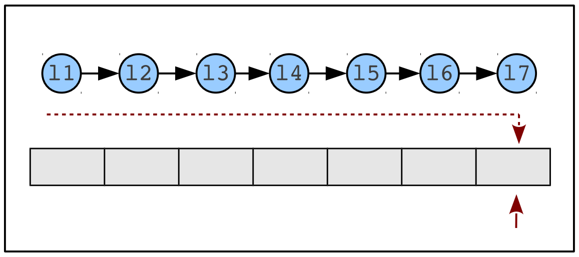

Figure 3. Search *l7* in Array

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*[1]. Items are then *unsafeCoerce*d on the way in and out based on the type information we keep in the heterogeneous list.

```
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 (v) of its stored element, are found with *ArrayFind*.

```
class ArrayFind r l v | r l → v where
  arrayFind :: r → l → 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.

```
hArrayGet :: ArrayFind r l v ⇒ ArrayRecord r → l → 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.

```
instance (HEq l l′ b
         , ArrayFind′ b v′ r l v n
         , ToValue n) ⇒
  ArrayFind (HCons (Field l′ v′) r) l v where
    arrayFind (HCons f r) l =
      toValue (arrayFind′ (hEq l (label f)) (value f) r l)
```

[1] A special type that can be used as a safe placeholder for any value.

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.

```
arrayFind′ :: ArrayFind′ b v′ r l v n
           ⇒ b → v′ → r → l → n
arrayFind′ = ⊥
data HZero
data HSucc n
class ArrayFind′ b v′ r l v n | b v′ r l → v n
instance ArrayFind′ HTrue v r l v HZero
instance (HEq l l′ b, ArrayFind′ b v′ r l v n)
  ⇒ ArrayFind′ HFalse v″ (HCons (Field l′ v′) r) l
              v (HSucc n)
```

The types *HZero* and *HSucc* implement naturals at type-level. If the label is found, then the index *HZero* is returned. Otherwise, we increase the index by one (*HSucc*) and continue searching. Once the index is found it has to be converted into an *Int* value, in order to use this value as the index of the array. This is done by the function *toValue*.

```
class ToValue n where
  toValue :: n → Int
```

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

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

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*.

```
instance ToValue HZero where
  toValue _ = 0
hPrev :: HSucc n → n
hPrev = ⊥
instance ToValue n ⇒ ToValue (HSucc n) where
  toValue n = 1 + toValue (hPrev n)
```

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.

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

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

```
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 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.

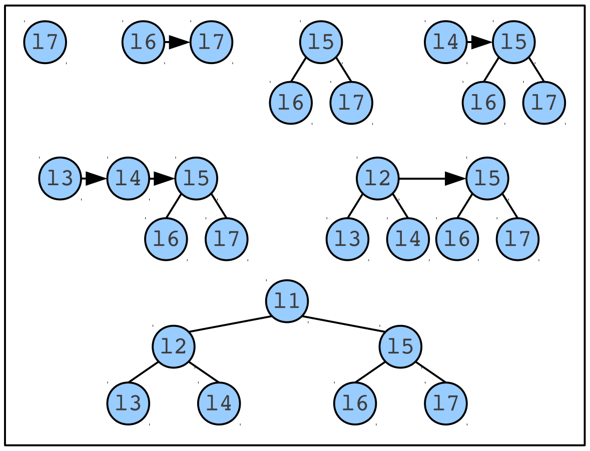

Figure 4. Insertion in a Skew

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
instance HHeight t h ⇒
         HHeight (HNode e t t') (HSucc 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 | l → 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' ts)) 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'
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 t (HCons t' r))
    (HCons (HNode f t t') r) where
  hSkewExtend' _ f (HCons t (HCons t' r)) =
    (HCons (HNode f t t') r)
```

3.3.2 Lookup

Now, we turn to the introduction of *HSkewGet*, which explores all paths at compile time but follows only the right one at run time.

```
class HSkewGet r l v | r l → v where
  hSkewGet :: r → l → v
```

Deciding on the path to the desired field is now more involved. The cases that both the test function and the worker function must consider are more numerous and long. Thus, we merge both functions. *HSkewGet* returns a type level and value level Maybe, that is, *HNothing* when no field with the label is found, and *HJust* of the field's type/value otherwise. For branching constructors *HCons* and *HNode*, *HPlus* (presented in subsection 2) chooses the correct path for us.

We will run *HSkewGet* on both the spine and each tree, so we have two base cases. *HNil* is encountered at the end of the spine, and *HEmpty* at the bottom of trees. In both cases, the field was not found, so we return *HNothing*.

```
instance HSkewGet HNil l HNothing where
  hSkewGet _ _ = HNothing
instance HSkewGet HEmpty l HNothing where
  hSkewGet _ _ = HNothing
```

The *HCons* case must consider that the field may be found on the current tree or further down the spine. A recursive call is made for each sub-case, and the results are combined with *HPlus*. If the field is found in the current tree, *HPlus* returns it, otherwise, it returns what the search down the spine did.

```
instance
  (HSkewGet r  l vr
  , HSkewGet r' l vr'
  , HPlus vr vr' v) ⇒
   HSkewGet (HCons r r') l v where
  hSkewGet (HCons r r') l =
    hSkewGet r l `hPlus` hSkewGet r' l
```

Observe that when doing $hSkewGet\ r\ l$ `hPlus` $hSkewGet\ r'\ l$ if the label is not present in r then the type system chooses the second instance of *HPlus* (*HPlus HNothing b b*). Thus, by lazy evaluation, the subexpression $hSkewGet\ r\ l$ is not evaluated since *hPlus* in that case simply returns its second argument.
The *HNode* case is a bigger version of the *HCons* case. Here three recursive calls are made, for the current field, the left tree, and the right tree. Thus two *HPlus* calls are needed to combine the result.

```
instance
  (HSkewGet f  l vf
  , HSkewGet r  l vr
  , HSkewGet r' l vr'
  , HPlus vf  vr vfr
  , HPlus vfr vr' v) ⇒
   HSkewGet (HNode f r r') l v where
  hSkewGet (HNode f r r') l =
    hSkewGet f l
      `hPlus` hSkewGet r l
        `hPlus` hSkewGet r' l
```

Finally, the *Field* case, when a field is found, is the case that may actually build a *HJust* result. As in *HListGet* for linked lists, *HEq* compares both labels. We call *HMakeMaybe* with the result of the comparison, and *HNothing* or *HJust* is returned as appropriate.

```
instance
  (HEq l l' b
  , HMakeMaybe b v m) ⇒
   HSkewGet (Field l' v) l m where
  hSkewGet f l =
    hMakeMaybe
      (hEq l (label f))
      (value f)
```

When we repeat the experiment at the end of subsection 2.2, but constructing a *SkewRecord* instead of an *HList*:

```
rSkew =
  (L1 .=. True    ) `hSkewExtend`
  (L2 .=. 9       ) `hSkewExtend`
  (L3 .=. "bla"   ) `hSkewExtend`
  (L4 .=. 'c'     ) `hSkewExtend`
  (L5 .=. Nothing) `hSkewExtend`
  (L6 .=. [4, 5]  ) `hSkewExtend`
  (L7 .=. "last"  ) `hSkewExtend`
  emptySkewRecord
lastSkew = hSkewGet rSkew L7
```

the resulting core code is:

```
lastSkewCore = case rSkew of
  HCons t1 _ → 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.

```
class HSkewUpdate l e r r' | l e r → r' where
  hSkewUpdate :: l → e → r → r'
```

We use the lookup operation *HSkewGet* to discriminate at type-level whether the field with the searched label is present or not in the skew list.

```
instance (HSkewGet r l m
         , HSkewUpdate' m l e r r') ⇒
         HSkewUpdate l e r r' where
  hSkewUpdate l e r =
    hSkewUpdate' (hSkewGet r l) l e r

class HSkewUpdate' m l e r r' | m l e r → r' where
  hSkewUpdate' :: m → l → e → r → r'
```

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

```
instance HSkewUpdate' HNothing l e r r where
  hSkewUpdate' _ l e r = r
```

In the other cases (i.e. when lookup results in *HJust v*) we call *hSkewUpdate* recursively on all subparts in order to apply the update when necessary. Because of the previous instance (when lookup returns *HNothing*), at run time recursion will not enter in those cases where the label is not present. We start the process in the spine.

```
instance
  (HSkewUpdate l e t t'
  , HSkewUpdate l e ts ts') ⇒
  HSkewUpdate' (HJust v) l e (HCons t ts)
                             (HCons t' ts')
   where
   hSkewUpdate' _ l e (HCons t ts) =
     HCons (hSkewUpdate l e t)
           (hSkewUpdate l e ts)
```

On a *HNode*, *hSkewUpdate* is recursively called on the left and right sub-trees as well as on the element of the node.

```
instance
  (HSkewUpdate l e e' e''
  , HSkewUpdate l e tl tl'
  , HSkewUpdate l e tr tr') ⇒
  HSkewUpdate' (HJust v) l e (HNode e' tl tr)
                             (HNode e'' tl' tr')
   where
   hSkewUpdate' _ l e (HNode e' tl tr) =
     HNode (hSkewUpdate l e e')
           (hSkewUpdate l e tl)
           (hSkewUpdate l e tr)
```

Finally, when we arrive to a *Field* and we know the label is the one we are searching for (because we are considering the case *HJust v*), we simply return the updated field.

```
instance
  HSkewUpdate' (HJust v) l e (Field l v) e
    where
      hSkewUpdate' _ l e e' = e
```

At run time, this implementation of *hSkewUpdate* 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.

```
class HSkewTail ts ts' | ts → ts' where
  hSkewTail :: ts → ts'
```

In Figure 5 we show an example of the possible cases we can find.

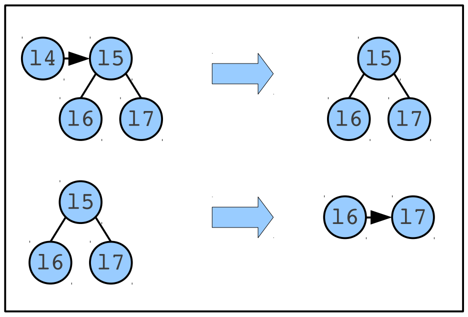

Figure 5. Tail in a Skew

The easy case is when the spine begins with a leaf. We just return the tail of the spine list.

$$\textbf{instance } \mathit{HSkewTail}\ (\mathit{HCons}\ (\mathit{HLeaf}\ e)\ \mathit{ts})\ \mathit{ts} \textbf{ where}$$
$$\quad \mathit{hSkewTail}\ (\mathit{HCons}\ \_\ \mathit{ts}) = \mathit{ts}$$

The other case is when the spine begins with a tree of three or more elements. Since *HLeaf* is a synonym of *HNode* with *HEmpty* as sub-trees, we need to assert the case when the sub-trees of the root *HNode* are nonempty (i.e. *HNode*s 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.

```
instance
  HSkewTail
    (HCons (HNode e t (HNode e' t' t'')) ts)
    (HCons t ((HCons (HNode e' t' t'')) ts))
  where
  hSkewTail (HCons (HNode _ t t') ts) =
    HCons t (HCons t' ts)
```

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

```
hSkewRemove l (HCons (HNode e t t') ts) =
  hSkewTail $
  hSkewUpdate l e (HCons (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 genertion (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 *ArrayRecord* and *SkewRecord* take the same time no matter the length of the record. Actually, sometimes larger records run faster than smaller records for *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 *ArrayRecord*, we disable the insertion optimization by immediately looking up the field just inserted. The insert-lookup process is run one million times. Only *ArrayRecord* is graphed because the other alternatives are too fast in this case. The graph exposes the linear time behavior of *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, *SkewRecord* is negligible. *HListRecord* is a linear graph picking up somewhat proba-

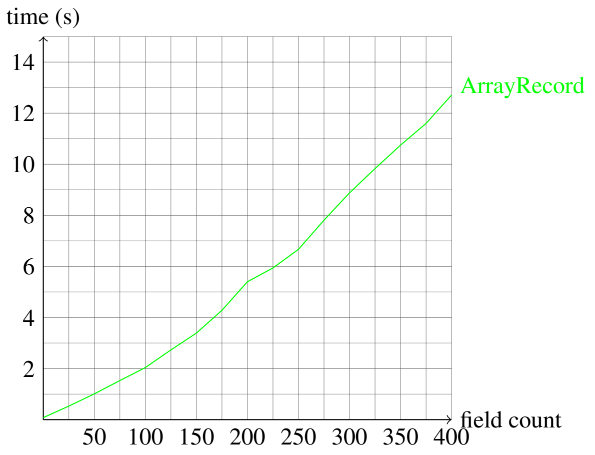

Figure 7. Extend: run time

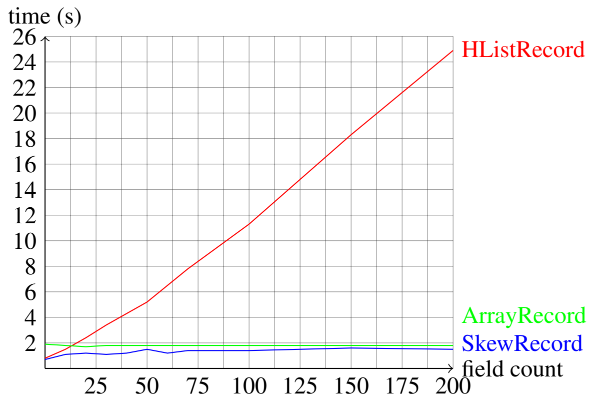

Figure 6. Lookup: run time

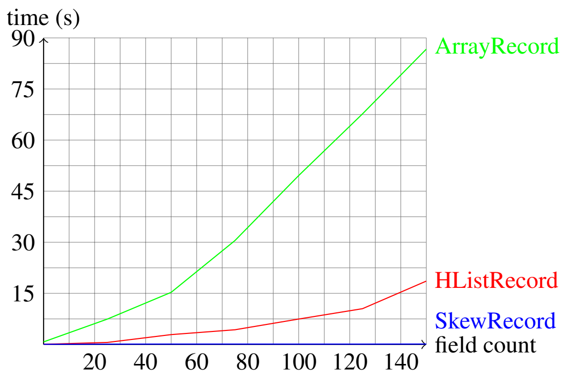

Figure 8. Update: run time

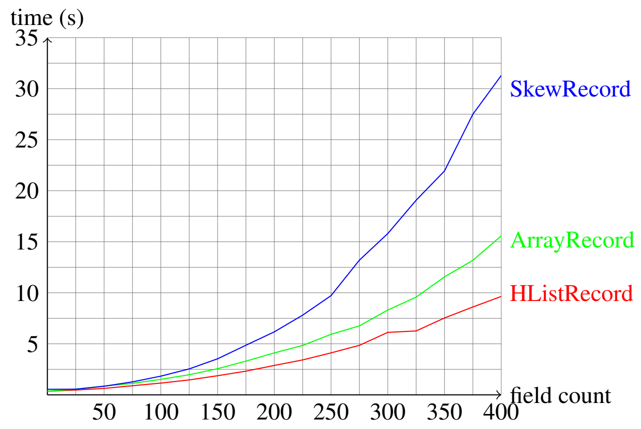

Figure 9. Lookup: compile time

bly after the CPU cache effects begins to play a role. *ArrayRecord* is also linear but much slower.

Figure 9 shows how compile time for the three implementations grows. *SkewRecord* is twice as slow as *HList* records, and *ArrayRecord* falls in between. When insertion is rare, we prefer *ArrayRecord* because of the compile time speed. Otherwise, *SkewRecord* is the best choice

5. Conclusions and Future Work

Using type level programming techniques we developed two new implementations of extensible records for Haskell: An array-like implementation, with constant time search and linear time insertion, and an impementation based on balanced trees that takes logarithmic time for searching and removing elements and constant time for inserting elements. This run time performance is achieved by moving most of the effort to compile time.

In the actual implementations we follow [15] in allowing label repetition. A type-predicate *HLabelSet* can be added to disallow this as in [13], with a slight cost in clarity but no cost in run time performance.

This approach can be used to improve the performance of systems that make extensive use of extensible records. Some examples of such systems are the first-class attribute grammars library AspectAG [24], the OOHaskell [12] library for object-oriented functional programming, or libraries for relational databases such as CoddFish [22] and HaskellDB [2].

Although the paper was focused on showing more efficient implementations of extensible records, our aim was mainly to show how harnessing type level programming techniques it is possible to improve the run time performance of some operations by moving certain computations to compile time. Type level programming is commonly used to increase the expressivity and type safety of programs, but in this paper we showed it can also be helpful for efficiency matters. This is the case specially for type level programming in Haskell, where there exists a phase distinction between compile and run time; types are computed at compile time while values are computed at run time.

Interesting future work is to find a way to reduce compilation time. Experiments demonstrate that GHC memoizes class instances, but some particularity of our instances seem to confuse the mechanism. [1] suggests constraint reordering and striving for tail calls to improve performance. It did not work for us and it made the presentation less clear, so we went with the straightforward version.

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

[1] Gershom Bazerman. Fixing performance leaks at the type level. URL `http://www.haskell.org/pipermail/haskell-cafe/2011-July/093974.html`.

[2] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. Student paper: HaskellDB improved. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 108–115. ACM Press, 2004.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *SIGPLAN Notices*, 40 (1):1–13, January 2005.

[5] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. NOTTCS-TR 96-3, Nottingham, 1996. URL `http://web.cecs.pdx.edu/~mpj/pubs/polyrec.html`.

[6] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1. ACM, 2008.

[7] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

[8] Rob Hoogerwoord. A logarithmic implementation of flexible arrays. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 191–207. Springer Berlin / Heidelberg, 1993.

[9] Wolfgang Jeltsch. Generic record combinators with static type checking. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 143–154. ACM, 2010.

[10] Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for haskell. In *Proceedings of the 1999 Haskell Workshop*, Paris, France, October 1999.

[11] Oleg Kiselyov. Type equality predicates: from OverlappingInstances to overcoming them, 2012. URL `okmij.org/ftp/Haskell/typeEQ.html`.

[12] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft, 2005.

[13] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[14] Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, December 2004.

[15] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[16] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17:241–248, 1983.

[17] Chris Okasaki. *Purely functional data structures*. PhD thesis, Pittsburgh, PA, USA, 1996. AAI9813847.

[18] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[19] Simon Peyton Jones and Greg Morrisett. A proposal for records in Haskell, 2003. URL `http://research.microsoft.com/en-us/um/people/simonpj/haskell/records.html`.

[20] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[21] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43(9):51–62, September 2008.

[22] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-489-8.

[23] Mark Tullsen. The zip calculus. In *In Fifth International Conference on Mathematics of Program Construction (MPC 2000*, pages 28–44. Springer-Verlag, 2000.

[24] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 245–256. ACM, 2009.