Just Do It While Compiling!
Fast Extensible Records in Haskell

Bruno Martínez    Alberto Pardo    Marcos Viera

Instituto de Computación
Universidad de la República
Montevideo - Uruguay
Record types

\[ \text{bruno} :: \{ \text{name :: String, surname :: String} \} \]

\[ \text{bruno} = \{ \text{name = "Bruno", surname = "Martinez"} \} \]
Record types

```
bruno :: { name :: String, surname :: String }
bruno = { name = "Bruno", surname = "Martinez" }
```

Selection

```
fullname p = p.name ++ " " ++ p.surname
```

Extension

```
bruno' = { phone = 27114244 | bruno }
```
Alternative record implementations

- **Tuple**: $O(1)$ selection, $O(n)$ extension
- **Linked list**: $O(n)$ selection, $O(1)$ extension
- **Search tree**: $O(\log n)$ selection, $O(\log n)$ extension
- **Hash table**: $O(1)$ selection, $O(1)$ extension
Our starting point is the Haskell library for strongly typed heterogeneous collections called **HList**.

- It provides an example implementation of extensible records.
- Labels are represented by singleton types.
- Search of a label in a record is performed at compile time.
- Selection in HList is linear time, insertion is constant time.
We propose two alternative implementations for extensible records in Haskell using the same techniques as HList.

- **ArrayRecord**, uses an array to hold the fields. 
  \(O(1)\) selection, but \(O(n)\) extension.

- **SkewRecord**, based on a balanced tree structure.
  \(O(\log n)\) selection, \(O(1)\) extension.

**Key idea:** We profit from the information given by the compile time search.

We do not worry to keep the work done at compile time superlinear if it helps us to speed up our programs at run time.
HList implements typeful heterogeneous collections using multi-parameter type classes and functional dependencies.

It strongly relies on *type-level programming* techniques where

- types represent type-level values,
- classes represent type-level functions.
class \(HBool\) \(b\)

\[HTrue;\] instance \(HBool\) \(HTrue\)

\[HFalse;\] instance \(HBool\) \(HFalse\)

\[hTrue = \bot :: HTrue\]

\[hFalse = \bot :: HFalse\]

class \(HNot\) \(t \to t'\) where

\[hNot :: t \to t'\]

instance \(HNot\) \(HFalse\) \(HTrue\) where

\[hNot \_ = hTrue\]

instance \(HNot\) \(HTrue\) \(HFalse\) where

\[hNot \_ = hFalse\]
class \textit{HList} \; l \\
\textbf{data} \; H\textit{Nil} \; = \; H\textit{Nil} \\
\textbf{data} \; H\textit{Cons} \; e \; l \; = \; H\textit{Cons} \; e \; l \\
\textbf{instance} \; \textit{HList} \; H\textit{Nil} \\
\textbf{instance} \; \textit{Hlist} \; l \Rightarrow \textit{HList} \; (H\textit{Cons} \; e \; l)
class HList l

data HNil = HNil
data HCons e l = HCons e l

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

exHList = HCons True (HCons 'a' HNil)
         :: HCons Bool (HCons Char HNil)
newtype Field l v = Field {value :: v}
(. = .) :: l → v → Field l v
_ . = . v = Field v
label :: Field l v → l
label = ⊥
**HList: extensible records**

\[
\textbf{newtype} \  Field \ l \ v = Field \ \{\ \text{value} :: v\ \}\n\]

\[
(. \ . \ = \ .) \ :: \ l \to v \to Field \ l \ v
\]

\[
_\ . \ . \ = \ . \ v = Field \ v
\]

\[
\text{label} :: Field \ l \ v \to l
\]

\[
\text{label} = \bot
\]

\[
\textbf{data} \ Name \ = \ Name
\]

\[
\textbf{data} \ Surname = Surname
\]

\[
\textbf{data} \ Phone \ = \ Phone
\]

\[
bruno = (Name \ . \ = \ ."Bruno") \ 'HCons' \\
(Surname \ . \ = \ ."Martinez") \ 'HCons' \\
(Phone \ . \ = \ .27114244) \ 'HCons' \\
HNil
\]
class HListGet $r$ $l$ $v$ | $r$ $l$ $→$ $v$ where
  hListGet :: $r$ $→$ $l$ $→$ $v$

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')) $v'$ $r'$ $l$

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 we write

\[
    phoneBruno = hListGet bruno Phone
\]

then GHC will transform it to the following definition in core:

\[
    phoneBrunoCore = \begin{array}{l}
        \text{case } bruno \text{ of} \\
        \quad \text{HCons } _b \rightarrow \text{case } b_1 \text{ of} \\
        \quad \quad \text{HCons } _b \rightarrow \text{case } b_2 \text{ of} \\
        \quad \quad \quad \text{HCons } p \rightarrow p
    \end{array}
\]
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

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

Any signals that `unsafeCoerce` is used to hide the actual type of values.
To find a field, we first search in the HList for the index and the type of the value.

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

Then with the index we extract the element from the array and cast it back to the correct type.

```
hArrayGet :: ArrayFind r l v ⇒ ArrayRecord r → l → v
hArrayGet (ArrayRecord r a) l = unsafeCoerce (a ! arrayFind r l)
```

Selection is then performed in constant time. Calculating the index is a compile time operation. Thanks to compiler inlining, the index is just a constant.
So far we have a structure with fast extension, list, and a structure with fast selection, array.

Naturally, the dictionary implementation in the middle is the search tree. Both selection and extension become log time.

<table>
<thead>
<tr>
<th>structure</th>
<th>selection</th>
<th>extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>

A search tree would require an ordering on the labels.

Types are not naturally ordered.
Unordered balanced trees

We don’t abandon trees yet, but we get rid of the ordering.

Because labels are static, when compiling a selection the compiler already knows where the field is.

The problem is then to find the adequate balanced tree structure.
A skew list is formed by a list of increasingly larger perfect binary trees.
SkewList example

- Only the first two trees are allowed to be of the same size.
- In that case, a new node becomes the parent of the two old trees.
- Otherwise, the new node is just inserted at the beginning of the list.
- The path to a node traverses the list (of logarithmic length) and down one tree (also logarithmic height), so selection is logarithmic.
SkewList example
SkewList example

```
2 → 1
```
SkewList example
SkewList example

Fast Extensible Records in Haskell
SkewList example

Fast Extensible Records in Haskell
SkewList example
SkewList example

![SkewList example diagram](image_url)
data HNode e l r = HNode e l r

data HLeaf e = HLeaf e

bruno =
  HLeaf (Name .==."Bruno") 'HCons'
  HNode (Surname .==."Martinez")
        (HLeaf (Phone .==.27114244 ))
        (HLeaf (Age .==.30 )) 'HCons'
  HNil
class *HSkewExtend* \( f \ r \ r' \mid f \ r \to r' \)

where \( hSkewExtend :: f \to r \to r' \)

class *HSkewGet* \( r \ l \ v \mid r \ l \to v \) where

\( hSkewGet :: r \to l \to v \)
Lookup (run time)

Field count

Time (s)

25 50 75 100 125 150 175 200

List

Array

Skew

Field count

Fast Extensible Records in Haskell
## Conclusion

<table>
<thead>
<tr>
<th>structure</th>
<th>selection</th>
<th>extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>skew list</td>
<td>$O(\log(n))$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>