Records with Rank Polymorphism

Justin Slepak
College of Computer Science
Northeastern University
USA
jrslepak@ccs.neu.edu

Olin Shivers
College of Computer Science
Northeastern University
USA
shivers@ccs.neu.edu

Panagiotis Manolios
College of Computer Science
Northeastern University
USA
pete@ccs.neu.edu

Abstract

In a rank-polymorphic programming language, all functions automatically lift to operate on arbitrarily high-dimensional aggregate data. By adding records to such a language, we can support computation on data frames, a tabular data structure containing heterogeneous data but in which individual columns are homogeneous. In such a setting, a data frame is a vector of records, subject to both ordinary array operations (*e.g.*, filtering, reducing, sorting) and lifted record operations—projecting a field lifts to projecting a column. Data frames have become a popular tool for exploratory data analysis, but fluidity of interacting with data frames via lifted record operations depends on how the language's records are designed.

We investigate three languages with different notions of record data: Racket, Standard ML, and Python. For each, we examine several common tasks for working with data frames and how the language's records make these tasks easy or hard. Based on their advantages and disadvantages, we synthesize their ideas to produce a design for record types which is flexible for both scalar and lifted computation.

CCS Concepts • **Software and its engineering** → **Polymorphism**; **Control structures**; **Data types and structures**; • **Mathematics of computing** → *Exploratory data analysis*.

Keywords array-oriented languages, rank polymorphism, data frames, records

ACM Reference Format:
Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2019. Records with Rank Polymorphism. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3315454.3329961

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARRAY '19, June 22, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6717-2/19/06…$15.00
https://doi.org/10.1145/3315454.3329961

1 Introduction

In a rank-polymorphic programming language, the primary control-flow mechanism is implicitly lifting functions to operate on array arguments of arbitrarily many dimensions. Instead of having the programmer write out a loop nest, the programmer effectively writes only the loop body, and the iteration structure is determined automatically based on the data being consumed. With iteration over the frame included in the semantics of function application, extracting an individual element is not the common way to consume aggregate data. Instead, the language favors uniform treatment of each element. This encourages the programmer to write in a style that depends less on the exact shape of the data, makes opportunities for parallelism more apparent to a compiler, and facilitates exploratory coding due to a lack of loop-nest boilerplate.

Record types are also aggregate data, but it is expected that different fields in a single record may have different types. Heterogeneity calls for the elements of a record to be treated individually rather than consuming such data through an analog of rank-polymorphic function lifting. Even when record fields have the same type, it is not generally appropriate to treat them uniformly—*e.g.*, temperature and latitude may both be stored as a number of degrees, but only one should be convertible between Fahrenheit and Celsius.

Libraries and even languages focused on "data frames," tabular data structures which are homogeneous within any column but potentially heterogeneous along a row, have arisen as popular tools for exploratory data analysis. We claim that data frames are the natural result of combining rank polymorphism with records.

Even with concise theoretical intuition about computing with data frames, ease of use often depends on things a theoretician might dismiss as uninteresting details. Our goal is to examine several designs for heterogeneous record data and how those designs fare when lifted up to data frames. In this work, we explore the interaction of records and rank polymorphism in settings where they are independent features We investigate what we get for free simply for doing record computation using rank polymorphism as it already exists without extra design work targeted at integrating the two features.

We demonstrate several common tasks using three programming systems' notions of records:

1. Racket's structs within `#lang remora/dynamic`, a rank-polymorphic DSL
2. Standard ML's records, with aggregate lifting provided by explicit use of `map`-like functions
3. Python's dictionaries, with a purpose-built library for working with data frames

After consideration of how each version of records helps or hinders exploratory analysis, we sketch a design for heterogeneous record data which combines their respective advantages for data frame manipulation but also serves as a sane design independent of our focus on data frames as an application domain.

2 Rank Polymorphism

In the rank-polymorphic programming model, the universe of data consists of regular arrays, whose "shape" is a sequence of natural-number dimensions. A function's definition includes the "ranks" of its arguments, *i.e.*, the number of dimensions each argument is expected to have. A polynomial evaluation function `poly-eval` would take a rank-1 argument containing the polynomial's coefficients and a rank-0 (*i.e.*, scalar) argument for the number at which to evaluate the polynomial. For a matrix inversion function `minv`, there would be a single argument of rank 2.

When a function is applied to arguments of higher rank, those arguments are viewed like nested arrays. Applying `minv` to an array with shape $3 \times 4 \times 4$ treats that argument as a 3-vector whose elements are 4×4 matrices. That 3-vector forms the implicit iteration space: we must invert each of the three matrices, producing a 3-vector of result matrices which we then reassemble into a $3 \times 4 \times 4$ result. The individual matrices in this scenario are called the "cells;" they are the fundamental unit on which the function operates. The "frame" is the sequence of leading dimensions in an argument's shape which drives the implicit iteration—appending the frame and cell shape gives the entire argument shape. Here, our 4×4-matrix cells are laid out in a 3-vector frame. We could also use this $3 \times 4 \times 4$ array as the coefficient argument to `poly-eval`, in which case it would be viewed as 4-vector cells in a 3×4-matrix frame.

Frames are compatible if and only if one is a prefix of the other. During function application, each argument's frame is extended to match the longest argument's frame, by replicating the argument cells. Continuing with `poly-eval` and the $3 \times 4 \times 4$ coefficient array, a 3-vector of values `[5 6 7]` would have to be grown to the 3×4 matrix

```
[[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]
```

Recall that the cell rank `poly-eval` expects for this argument is 0. So each *scalar* must be replicated on its own 4 times. Suppose instead we gave `[5 6 7]` as an argument to a `dot-prod` function which expects two rank-1 arguments. We would then have a scalar frame containing a single cell. If the other argument's shape is 2×3, we must grow our scalar frame into a 2-vector frame by replicating the entire *vector*—the cell rank here is 1 instead of 0 in the previous case. So the replicated form is `[[5 6 7] [5 6 7]]`.

The rank which a function expects for that argument effectively determines whether the matrix produced by frame expansion has the original vector as a column or as a row because the decomposition of the argument shape into its frame and cell portions determines where new dimensions are added. Viewing an n-vector as a vector frame containing scalar cells means it can be grown to an $n \times m \ldots$ array, whereas viewing it as a scalar frame containing a vector cell means it can be grown to an $m \ldots \times n$ array.

A common way of manipulating the iteration space is choosing where in an argument shape new dimensions can be added. Consider a simple vector-matrix addition:

```
(+ [10 20]
   [[1 2]
    [3 4]])
```

The function `+` expects scalar arguments. When we split the actual arguments' shapes—`[2]` and `[2 2]` respectively—into frame and cell portions, they are *frame* = `[2]` and *cell* = `[]` for the first argument and *frame* = `[2 2]` and *cell* = `[]` for the second. We must extend the first argument's `[2]` frame to `[2 2]` by replicating scalar cells. So the `10` cell expands to `[10 10]` and `20` to `[20 20]`. The first argument is therefore treated like `[[10 10] [20 20]]`. If we instead used a `vec+` function which expects vector arguments, the frame shapes would be `[]` and `[2]`. Expanding `[]` to `[2]` would replicate the *vector* cell `[10 20]` rather than each scalar cell. So we treat the first argument as `[[10 20] [10 20]]`.

The behavior of `vec+` could be described as breaking its arguments into vector cells and then applying `+` to those vectors (which internally breaks the vectors into scalar cells to add pointwise). So `vec+` is a "reranked" version of `+`. Reranking is a common enough technique in rank-polymorphic programming to warrant some language-level support, such as J's `"` operator.

2.1 A Rank-Polymorphic Language

In order to demonstrate use of rank polymorphism we will use a Racket-based DSL called `#lang remora/dynamic`. It is a prototype implementation of a dynamically typed variant of Remora [11], a higher-order, rank-polymorphic programming language. The syntax for this DSL is largely inherited from Racket. The two major differences are nested bracket notation for array construction and rank annotations on formal parameters.

An array literal can be written out with an explicit shape, such as this 2×3 matrix:

```
> (define arr1 (alit (2 3) 3 4 5 6 7 8))
```

It can also be written using nested brackets. Brackets around a sequence of expressions builds an array with one more dimension than those expressions' results. If `xs`, `ys`, and `zs` are each 4×2 arrays, then `[xs ys zs]` is a $3 \times 4 \times 2$ array. Since a bracketed list is itself an expression, it can appear inside another bracketed list. While there is still a conceptual distinction between expressions and atoms, `#lang remora/dynamic` implicitly promotes any syntactic atom appearing where an expression is expected to a scalar array with that as its sole atom. So even though numerals are syntactically atoms, they can be used as function arguments, returned values, or bracketed-array components. Combining these notational conveniences, we have another way to write out the same matrix shown above as a literal:

```
> (define arr2 [[3 4 5] [6 7 8]])
> (equal arr1 arr2)
#t
```

When writing out a function, whether with `define` or λ, each formal parameter is marked with the rank of the cells into which its argument should be divided during function application. Within the function body, that variable is bound to an array of that rank. Applying this polynomial evaluation function will treat the first argument as containing vector cells and the second argument as containing scalar cells. The use of the variable `coeffs` can only mean a vector, but a single application of `poly-eval` might repeat this many times with `coeffs` referring to different vectors.

```
> (define (poly-eval (coeffs 1) (x 0))
    (reduce + 0
            (* coeffs
               (expt x
                     (iota [(length coeffs)])))))
```

Any natural number is a valid cell-rank specifier, but a formal parameter can also be given the rank `all`, indicating that the entire argument should be treated as a single cell. The `all` rank is typically used in layout-manipulating operations, like `rotate` and `reverse`, as well as reduction and scan operations operating along an array's major axis. For example, this `sum` function computes the sum along its argument's major axis, turning an array with shape `[m n ...]` into one with shape `[n ...]`.

```
> (define (sum (x all))
    (reduce + 0 x))
```

If we want to sum an array along some other axis, we could do so with a reranked version of `sum`. There is syntactic sugar for reranking in `#lang remora/dynamic`: Preceding a function with a ~ and a parenthesized list of argument ranks produces the corresponding η-expansion. So `~(2)sum` expands (with a fresh variable `a`) to

```
(λ ((a 2)) (sum a))
```

Using `sum` along the major axis of the 2×3 array `arr1` produces a 3-vector result. Asking for the `sum` along the "rank-2 axis" changes nothing, since that is already `arr1`'s major axis. With `~(1)sum`, reranking `sum` to use the rank-1 axis, we apply the `sum` function to each individual vector within `arr1` and get two result cells, assembled in a vector frame.

```
> (sum arr1)
[9 11 13]
> (~(2)sum arr1)
[9 11 13]
> (~(1)sum arr1)
[12 21]
```

It is necessary to have an escape hatch from working entirely with regular data. This necessity may arise from raw input data itself, such as a collection of strings of non-uniform length. Another potential cause is that certain array operations produce output whose shape depends on input values (not just the input shape). Lifting such an operation can produce result cells of differing shape, which must somehow be assembled into the same frame. Since cells of differing shape cannot coexist on their own, we use a "box" as a wrapper. A box is an atom, but it contains an arbitrary array. So we can produce ragged data when necessary:

```
> (define ragged
    [(box [1 2])
     (box [3 4 5])])
```

Consuming boxed data requires unpacking the box's contents, which is done like let-binding. The let-bound contents can then be consumed like an ordinary array.

```
> ((λ ((some-box 0))
     (unbox contents some-box
       (sum contents)))
   ragged)
[3 12]
```

However, if the computation we intend to perform on the box's contents has a result shape dependent on input shape, it is safest to produce boxed output, in case we happen to be lifting over an array of many boxes.

```
> ((λ ((some-box 0))
     (unbox contents some-box
       (reverse contents)))
   ragged)
 Result cells have mismatched shapes
> ((λ((some-box 0))
     (unbox contents some-box
       (box (reverse contents))))
   ragged)
[(box [2 1])
 (box [5 4 3])]
```

2.2 Heterogeneous Data

Rank polymorphism facilitates uniform treatment of homogeneous data, but not heterogeneous data such as tuples or records. Converting a series of temperature readings from Fahrenheit to Celsius makes sense, but not a log entry containing a city name, temperature, and time of day. Although heterogeneous data calls for heterogeneous treatment, it is still common to work with regular arrays whose elements themselves are all of one particular type of heterogeneous data. Our hypothetical weather report may be a single item in a large table. Such use cases are common enough to have spawned several popular programming systems—such as R [10] and Pandas [8]—for working with "data frames," rectangular tables which are heterogeneous along the row axis but homogeneous within any single column. Typical operations on a data frame include:

1. Constructing a data frame from a collection of rows or columns
2. Extracting an individual column
3. Transforming data in a column
4. Selecting the subset of rows which fit some predicate, such as certain column values
5. Grouping rows into separate tables by partitioning a column's values
6. Summarizing grouped rows

Although we have settled on a fixed notion of how to operate on array data, when we envision data frames as arrays of records (treating records as atoms from rank polymorphism's perspective) ergonomic convenience of the above operations depends heavily on how records themselves are consumed and produced. Since Remora is statically typed, we would like to keep an eye towards typability in sketching a design for records.

3 Candidate 1: Racket-style Structure

In `#lang remora/dynamic`, we have access to quite a lot of Racket's built-in machinery, including `struct`s, the preferred form of record data. We can define individual table rows as `struct`s, but we must first define the `struct` type we intend to use:

```
> (struct weather (loc day month year hi lo))
```

This structure-type declaration defines `weather` as a function which produces a structure when given the field values:

```
> (define dallas-temp
    (weather "Dallas" 28 3 2019 74 57))
```

It also defines functions `weather-loc`, `weather-day`, *etc.*, which we can use to access individual `struct` fields.

```
> (weather-loc dallas-temp)
"Dallas"
> (weather-year dallas-temp)
2019
> (weather-lo dallas-temp)
57
```

3.1 Table Creation

We can build a data frame as a vector containing several `weather` `struct`s.

```
> (define temp-readings
    [dallas-temp
     (weather "Dublin"  1 4 2019 11  5)
     (weather "Nome"   31 3 2019 31 26)
     (weather "Tunis"  31 3 2019 21 12)])
```

This table could also have been built from columns instead of rows. Since `weather` is a function expecting six scalar arguments, it can be applied to vector arguments to get a vector of `struct`s.

```
> (define temp-readings
    (weather
     ["Dallas" "Dublin" "Nome" "Tunis"]
     [28 1 31 31]
     [3 4 3 3]
     2019
     [74 11 31 21]
     [57 5 26 12]))
```

Note that the year argument we gave is still scalar, meaning all of our temperature readings are from the same year. When function application lifts all arguments to have the same frame shape, the scalar `2019` is promoted to a vector with `2019` for every atom.

3.2 Column Extraction

In the same way that the constructor function `weather` lifts over arrays of field values, field-accessor functions lift over arrays of `struct`s:

```
> (weather-loc temp-readings)
["Dallas" "Dublin" "Nome" "Tunis"]
> (weather-lo temp-readings)
[57 5 26 12]
```

Although we constructed this data frame with a scalar as one of the column arguments, we still have a vector result when we extract that column.

```
> (weather-year temp-readings)
[2019 2019 2019 2019]
```

This method of column extraction gives the same result for either definition of `temp-readings`. Whether the data frame was constructed as a vector of row values or by lifting the constructor over the intended columns, the resulting array value is the same.

If we decide to focus our investigation on recorded high temperatures, it might be convenient to have a table which elides the lows. Here we hit a snag. By reranking a vector of field accessors, we can build an array—but not a proper data frame—containing the information we want.

```
> (~(0)[weather-loc weather-day
        weather-month weather-hi]
    temp-readings)
[["Dallas" 28 3 74]
 ["Dublin" 1 4 11]
 ["Nome" 31 3 31]
 ["Tunis" 31 3 21]]
```

Here we have a rank-2 array, a table whose rows are heterogeneous vectors. A design which requires *vectors* to support differing atom types is undesirable for performance reasons and difficult to build a suitable type system for.

Producing a vector of structs—which must have a different number of fields since we are eliding some of weather's fields—requires defining a new struct type.

```
> (struct (weather-v2 loc day month hi))
```

We then have a more awkward transformation function.

```
> (define hi-readings
    ((λ ((w 0))
       (weather-v2 (weather-loc w)
                   (weather-day w)
                   (weather-month w)
                   (weather-hi w))
     temp-readings))
```

This could be more convenient with a composition form which preprocesses each argument with a unary function before passing them all along to a final higher-arity function. We will use the notation

```
(comp* f g ...)
```

as syntactic sugar for

```
(λ ((x all) ...) (f (g x) ...))
```

with a fresh x for each g.

```
> (define hi-readings
     ((comp* weather-v2
       weather-loc weather-day
       weather-month weather-hi)
      temp-readings))
```

3.3 Column Update

A Racket struct type definition does not automatically create field-update functions (except for fields designated as mutable), though such functions could be defined:

```
> (define (weather-lo-set w l)
    (weather (weather-loc w)
             (weather-day w)
             (weather-month w)
             (weather-year w)
             l
             (weather-hi w)))
```

If we plan to eventually integrate this data into a larger multi-year dataset, with a year column indexed from the start date of data collection, we may need to replace 2019 with some particular offset from the start year.

```
> (define year-adjusted
    (weather-year-set temp-readings offset))
```

Looking over our reported temperatures, something seems off: Why is Tunis colder than Nome? Then it dawns on us that our reporting stations in the USA must have given temperatures in Fahrenheit. As long as we can recognize which locations are in the USA, presumably using an in-usa? predicate, we can write a function to normalize temperature data[1].

```
> (define (f->c (t 0))
    (/ (- t 32) 1.8))
> (define (normalize-temps (w 0))
    (select (in-usa? (weather-loc w))
            (weather
             (weather-loc w)
             (weather-day w)
             (weather-month w)
             (weather-year w)
             (f->c (weather-hi w))
             (f->c (weather-lo w)))
            w))
```

Updating a column conditionally according to values in other columns can then use the same implicit control flow.

```
> (normalize-temps temp-readings)
[(weather "Dallas" 28 3 2019 23.33 13.89)
 (weather "Dublin" 1 4 2019 11 5)
 (weather "Nome" 31 3 2019 -0.56 -3.33)
 (weather "Tunis" 31 3 2019 21 12)]
```

3.4 Row Filter

The filter function takes two arguments: a boolean vector to use as a mask and an array whose leading axis has the same length as that boolean vector. The result array includes the sub-arrays from the second argument in positions corresponding to the boolean vector's true entries. This is in contrast to the usual functional filter, which takes a predicate function as its first argument; it is rather a specialization of the "replicate"/"compress" function from APL tradition.

[1]In the interest of keeping control flow regular, conditionals in Remora use a boolean value to select between two possible result values, which are both computed eagerly.

Suppose we want to look at only the temperature reports from Dublin.

```
> (define (from-dublin? (w 0))
    (string=? "Dublin" (weather-loc w)))
> (define d (from-dublin? temp-readings))
> d
[#f #t #f #f]
> (filter d temp-readings)
[(weather "Dublin" 1 4 2019 11 5)]
```

If the order of entries in the table is unimportant, we now have an alternative strategy for fixing our temperature scale disparity. Since the `temp-readings` table is a vector of `structs`, we can isolate the rows with USA locations:

```
> ((compose in-usa? weather-loc) temp-readings)
[#t #f #t #f]
> (filter ((compose in-usa? weather-loc)
           temp-readings)
   temp-readings)
[(weather "Dallas" 28 3 2019 74 57)
 (weather "Nome" 31 3 2019 31 26)]
> (filter (not ((compose in-usa? weather-loc)
                temp-readings))
   temp-readings)
[(weather "Dublin" 1 4 2019 11 5)
 (weather "Tunis" 31 3 2019 21 12)]
```

Using the boolean mask which selects reports from sources in the USA, we can put their Celsius-converted temperatures into a new table with the unaltered non-USA reports.

```
> (append
   (normalize-temps
    (filter ((compose in-usa? weather-loc)
             temp-readings)
            temp-readings))
   (filter (not ((compose in-usa? weather-loc)
                 temp-readings))
            temp-readings))
[(weather "Dallas" 28 3 2019 23.33 13.89)
 (weather "Nome" 31 3 2019 -0.56 -3.33)
 (weather "Dublin" 1 4 2019 11 5)
 (weather "Tunis" 31 3 2019 21 12)]
```

3.5 Row Partition

Generalizing from the previous example, it is useful to have a suite of ways to break up a table according to the values in its rows. A function for splitting a table into rows that match a predicate and rows that don't will often produce ragged data, so the individual partitions of the table must be wrapped as boxes. We have a `filter*` function, meant to be a version of `filter` which is safe for lifting over several different boolean vectors. While applying `filter` will raise a dynamic error if it is lifted over boolean vectors with different numbers of `#t` elements, `filter*` keeps each result cell in a box.

```
> (filter [#t #f #t] [1 2 3])
[1 3]
> (filter [[#t #f #t] [#f #t #f]] [1 2 3])
```
Result cells have mismatched shapes
```
> (filter* [[#t #f #t] [#f #t #f]] [1 2 3])
[(box [1 3]) (box [2])]
```

We can construct our two boolean vectors for identifying USA and non-USA temperature reports. Note that `[id not]` is a vector of scalar functions. If it appears in function position, it contributes its own frame, `[2]`, to the function application's overall frame. This is our desired frame, but it is incompatible with the frame of the temperature readings table (which is some vector of length much more than 2). We need this `to-all` function as an adapter. It converts each individual function to a function which consumes its entire argument as one cell (before then passing it to the underlying function).

```
> (define (to-all (f 0))
    (λ ((a all)) (f a)))
```

When we apply `(to-all [id not])`, we have a vector of `all`-ranked functions, so the argument is considered to have a scalar frame. That means the argument will be passed in its entirety to each of `id` and `not`.

```
> (define in-or-out
    ((to-all [id not])
     ((compose in-usa? weather-loc)
      temp-readings)))
> in-or-out
[[#t #f #t #f]
 [#f #t #f #t]]
```

So a general `split` function can be written as

```
> (define (split (pred 0) (tbl all))
    (filter* ((to-all [id not]) (pred tbl))
             tbl))
> (split (compose in-usa? weather-loc)
         temp-readings)
[(box [(weather "Dallas" 28 3 2019 74 57)
       (weather "Nome" 31 3 2019 31 26)])
 (box [(weather "Dublin" 28 3 2019 74 57)
       (weather "Tunis" 31 3 2019 31 26)])]
```

With `filter*` able to lift over multiple boolean vectors, we can consider other ways to generate collections of boolean vectors to partition the table. The core of the task is to generate the vector of predicates, which can then be applied to each element in the table.

A numeric column might be partitioned into ranges. We can build each range's predicate by combining the upper- and lower-bound predicates:[2]

[2]Here we borrow Racket's `and/c` combinator, which works like `and`, but lifted to operate on predicates.

```
> (define (bins (bounds 1))
    (and/c (append [(const #t)]
                   ((curry <) bounds))
           (append ((curry >=) bounds)
                   [(const #t)])))
```

Then filtering the vector [1 3 5 7 9 2 4 6 8] according to the result from (bins [5 10]) gives the following partitioned results:

```
[(rem-box [1 3 5 2 4])
 (rem-box [7 9 6 8])
 (rem-box [])]
```

We might also want to partition our table so as to group temperature readings from the same location. This requires identifying the set of unique locations mentioned in the table and then constructing an equality predicate matching each of them.

```
> (define (uniques (v 1))
    (to-all
     ((curry equal?) (nub v))))
```

The nub function condenses a vector down to a vector of its unique elements. So the result from uniques is a vector of predicates, each of which checks whether its argument is equal to the corresponding element of the original vector. If we generate this collection of predicates from our table's location column and then apply them to that column, we get a collection of boolean vectors each of which indicates which rows have a particular location. So filter* will then build a list of tables, with the rows grouped according to their location fields.

3.6 Ergonomics

We get quite a lot of flexibility in data manipulation by combining orthogonal features: rank polymorphism and traditional array-manipulation operators with Racket's built-in functional composition combinators and struct types. Constructing tables from columns and computing on specific columns both fall out of making Racket's pre-existing struct constructors and field accessors rank-polymorphic. Filtering rows according to a predicate lifts to grouping rows by using several non-overlapping predicates.

Some of the awkward code here arises from Racket's somewhat clumsy point-free programming. Code for manipulating data frames often calls for unary predicates and transformation functions, which we build by partially applying and adding plumbing between already-existing functions. Idiomatic Racket code mostly avoids point-free programming, in contrast to Haskell or J. In principle, fixing this should only require a combinator library better suited for concise point-free code.

The more persistent problem with using Racket structs as data-frame rows is in tying each struct operation to a particular declared type. Every combination of column names must be declared and named before use, and every column selection must give that combination's name as well.

4 Candidate 2: Standard ML-style Record

One alternative design for labeled heterogeneous data which avoids some of the shortcomings of Racket's structs is Standard ML's records [9]. In Standard ML, a record type consists of a (statically specified) set of field names and a required type for each field. In contrast to Racket's struct types, Standard ML does not require record types themselves to be named. However, Standard ML lacks Remora's automatic lifting over argument frames, so producing working code requires sprinkling explicit map and replication operations throughout. In the interest of keeping the focus on records, rather than on *ad hoc* implementations of rank polymorphism, we will elide the definitions of most of these utility functions. Past work has considered automatically generating the required map and replication operations for a substantial subset of cases handled by rank polymorphism [12].

4.1 Table Creation

Our example row from Section 3 can be written as

```
- val dallas_temp = {loc="Dallas", day=28,
    month=3, year=2019, hi=74, lo=57}
```

As in Racket, several rows can be packed into a list, but we do not have a direct way to construct a table from columns. Record creation in Standard ML is special syntax rather than a function, so we cannot map it over lists of column values without explicitly defining a record-creation function and a map variant of appropriate arity.

```
- fun weather(lc,d,m,y,h,l) =
    loc=lc, day=d, month=m, year=y, hi=h, lo=l;
- val temp_readings = map6 weather
    (["Dallas","Dublin","Nome","Tunis"],
     [28, 1, 31, 31],
     [3, 4, 3, 3]
     [2019, 2019, 2019, 2019]
     [74, 11, 31, 21]
     [57, 5, 26, 12])
```

The list repeating 2019 is due to the rank-monomorphic nature of an explicit map. If we were to name our columns before assembling them into a table, we could instead (explicitly) construct the repeated list:

```
- fun rep n x = if (n <= 0) then []
    else x::(rep (n - 1) x);
val rep = fn : int -> 'a -> 'a list
- val cities = ["Dallas", "Dublin",
    "Nome", "Tunis"];
...
- val years = rep (length cities) 2019;
```

Or alternatively

```
- val years = map (fn _ => 2019) cities;
```

4.2 Column Extraction

We still have easy access to an individual column in Standard ML because record accessors are first-class functions.

```
- map #hi temp_readings;
val it = [74,11,31,21] : int list
```

However, assembling a subset of columns still requires writing a row constructor for the new field set because it is another case of constructing a table from columns.

```
- fun hi_temp (c,h) = loc=c,hi=h;
val hi_temp = fn : 'a * 'b -> hi:'b, loc:'a
- map2 hi_temp (map #loc temp_readings,
    map #hi temp_readings);
val it =
  [hi=74,loc="Dallas",
   hi=11,loc="Dublin",
   hi=31,loc="Nome",
   hi=21,loc="Tunis"]
  : hi:int, loc:string list
```

A slight awkward point is that field access functions cannot be typed in isolation. If a field access function does not appear in a context which constrains the particular field set of the record it will be applied to, the function does not have a single most general Standard ML type. This might impede a user who prefers to build up a collection of defined data-manipulation gadgets for later reuse.

4.3 Column Update

As in Racket, we lack pre-existing functions for updating individual fields, and when we write our own, they are specific to the set of fields in the resulting record. For example, this adaptation of the previous section's temperature normalization function only works on records with the same set of field names as our `temp_reports` table.[3]

```
- fun normalize_temps w =
    if in_usa (#loc w) then
      {loc = #loc w,
       day = #day w,
       month = #month w,
       year = #year w,
       hi = ((#hi w) - 32.0)/1.8,
       lo = ((#lo w) - 32.0)/1.8}
    else w;
```

4.4 Row Filter

Standard ML includes a `filter` function, so we can still easily select rows which match a predicate. The difficulty introduced by Standard ML's type system is that predicates must specify which fields they ignore, not just the fields they inspect, in order to be typable.

[3]It also does not work on `temp_reports` itself due to numeric type incompatibility, but this is not to be blamed on Standard ML's *record* types.

```
- fun from_dublin w = #loc w = "Dublin";
Error: unresolved flex record
   (can't tell what fields there are
    besides #loc)
- fun from_dublin
   {loc = l, day = _, month = _, year = _,
    hi = _, lo = _} =
   l = "Dublin";


val from_dublin = fn
  : {day:'a, hi:'b, lo:'c, loc:string,
    month:'d, year:'e} -> bool
- List.filter from_dublin temp_readings;
val it = [{day=1,hi=11,lo=5,loc="Dublin",
           month=4,year=2019}]
  : {day:int, hi:int, lo:int, loc:string,
     month:int, year:int} list
```

In principle, this restriction exists with Racket-style structs as well: each field accessor is tied to a particular struct type.

4.5 Row Partition

As in filtering, partitioning is as straightforward as is allowed by Standard ML's field-set monomorphism. Again, Standard ML's lack of rank polymorphism requires extra code to adapt the `filter` function to an aggregate first argument and singular second argument, but once that adapter is available, the table can be partitioned using a list of disjoint predicates.

```
- fun many2one f xs y = map (fn x => f x y) xs;
val many2one = fn
  : ('a -> 'b -> 'c) -> 'a list -> 'b -> 'c list
- many2one List.filter
   [in_usa o #loc, not o in_usa o #loc]
   temp_readings;
val it =
  [[{day=28,hi=74,lo=57,loc="Dallas",
     month=3,year=2019},
    {day=31,hi=31,lo=26,loc="Nome",
     month=3,year=2019}],
   [{day=1,hi=11,lo=5,loc="Dublin",
      month=4,year=2019},
    {day=31,hi=21,lo=12,loc="Tunis",
       month=3,year=2019}]]
  : {day:int, hi:int, lo:int, loc:string,
     month:int, year:int} list list
```

Standard ML's ubiquitous currying makes the predicate-list construction functions more syntactically lightweight, for example:

```
- fun split pred table =
    many2one List.filter
     [pred, not o pred] table;
val split = fn
    : ('a -> bool) -> 'a list -> 'a list list
- split from_dublin temp_readings;

val it =
  [[day=1,hi=11,lo=5,loc="Dublin",
     month=4,year=2019],
   [day=28,hi=74,lo=57,loc="Dallas",
      month=3,year=2019,
    day=31,hi=31,lo=26,loc="Nome",
      month=3,year=2019,
    day=31,hi=21,lo=12,loc="Tunis",
      month=3,year=2019]]
  : {day:int, hi:int, lo:int, loc:string,
     month:int, year:int} list list
```

4.6 Ergonomics

With first-class record types, it is not strictly necessary to declare every set of columns we will use in some table. We do not have to worry about whether a row we want to add into a table has its fields in the right order because record type equality is independent of field ordering. This is in contrast to Racket-style structs, where all fields are positional (and their names are only used for projection functions). Functions like `filter` and `map` still work as expected (though with more needed adapter code than in a rank-polymorphic language) because these are core functional programming tools, not something specific to array-oriented languages or libraries.

However, we lose some of the flexibility that might be expected from the set-of-names style of record. Despite having first-class record types, Standard ML echoes many of the ergonomic faults of Racket's structs. Record types can be used without declaration, but we don't get a liftable constructor function without explicitly writing one. It is also awkward to build up a task-specific toolkit because field accessor functions cannot be used without somehow specifying the full set of *other* field names. Exploratory analysis using only a subset of the original columns would require redefining filtering, partitioning, and transformation functions originally written for the full table. Many of the problems requiring repetitive per-type solutions could be eliminated by adding row polymorphism, at the cost of complicating type inference [13].

5 Candidate 3: Python-style Dictionary

Python's structure for labeled heterogeneous data is a dictionary mapping strings to field values. Unlike in Racket and Standard ML, field names themselves are run-time values, allowing arbitrary string computation to choose which field to project out. Dictionary construction syntax is similar to Standard ML's record construction.

```
>>> dallas_temp = {'loc': 'Dallas', 'day': 28,
     'month': 3, 'year': 2019,
     'hi': 74, 'lo': 57}
```

Typical practice for working with tabular data in Python is to not build tables directly from the language's primitive data structures but to use the Pandas library [8]. Pandas stores a data frame as a collection of labeled columns, in contrast to previous sections' use of collections of rows. While the list-of-rows representation is also possible in Python, the sort of helper functions needed for emulating rank polymorphism in order to work with that representation fluidly are generally considered non-idiomatic Python.

5.1 Table Creation

As before, several rows can be packed into a list, and that list can be used to build a Pandas `DataFrame` object.

```
>>> temp_list =
  [dallas_temp,
   {'loc': 'Dublin', 'day': 1, 'month': 4,
    'year': 2019, 'hi': 11, 'lo': 5},
   {'loc': 'Nome', 'day': 31, 'month': 3,
    'year': 2019, 'hi': 31, 'lo': 26},
   {'loc': 'Tunis', 'day': 31, 'month': 3,
    'year': 2019, 'hi': 21, 'lo': 12}]
>>> by_rows = pd.DataFrame(temp_list)
```

Alternatively, a `DataFrame` can be constructed from a dictionary of columns.

```
>>> by_cols =
  pd.DataFrame({'loc': ['Dallas', 'Dublin',
                        'Nome', 'Tunis'],
    'day': [28, 1, 31, 31],
    'month': [3,4,3,3],
    'year': 2019,
    'hi': [74, 11, 31, 21],
    'lo': [57,5,26,12]})
```

5.2 Column Extraction

Since a Pandas DataFrame is kept as a collection of columns, extracting one column is a simple indexing operation, with no need for implicit or explicit mapping. Each individual column is represented as a `Series` object, which can be consumed using list comprehensions but also has a `map` method for unary functions.

```
>>> by_cols['loc']
0    Dallas
1    Dublin
2      Nome
3     Tunis
```

The indexing operator can lift to extract multiple columns by passing a list of column names, producing a `DataFrame` rather than a simple list of `Series`.

```
>>> by_cols[['loc','hi']]
      loc  hi
0  Dallas  74
1  Dublin  11
2    Nome  31
3   Tunis  21
```

5.3 Column Update

Dropping in a replacement column requires no lifting, for the same reason as column extraction. A column can also still be updated by applying a transformation function to each row. This would be a straightforward list comprehension in a table represented as a list of rows. Otherwise a `DataFrame`-specific method must be used with the axis explicitly specified, rather than the language's native machinery (iterating over a data frame itself steps through its columns).

```
>>> def f2c(f): return (f - 32)/1.8
>>> def normalize_temp(r):
...     if in_usa(r['loc']):
...             r['lo'] = f2c(r['lo'])
...             r['hi'] = f2c(r['hi'])
...     return r
>>> by_cols.apply(normalize_temp, axis=1)
      loc  day  month  year     hi     lo
0  Dallas   28      3  2019  23.33  13.89
1  Dublin    1      4  2019  11.00   5.00
2    Nome   31      3  2019  -0.56  -3.33
3   Tunis   31      3  2019  21.00  12.00
```

Both the list-of-dictionaries representation and Pandas also offer horizontal concatenation of `DataFrames`, which would have to be reimplemented for each pair of column sets using Racket structs or Standard ML records.

5.4 Row Filter

A `DataFrame`'s `loc` method can take a boolean mask, similar to Remora's `filter` function. Our first example from Section 3 of selecting only readings from Dublin can be written easily by constructing a `Series` with `True` in positions corresponding to rows where our table's `'loc'` field is `'Dublin'`.

```
>>> by_cols['loc'] == 'Dublin'
0    False
1     True
2    False
3    False
>>> by_cols[by_cols['loc'] == 'Dublin']
      loc  day  month  year  hi  lo
1  Dublin    1      4  2019  11   5
```

However, the implicit lifting we see for == is special treatment given only to certain primitives. A user-written predicate, like our imagined `in_usa` check, requires explicit lifting.

```
>>> [in_usa(l) for l in by_cols['loc']]
[True, False, True, False]
>>> by_cols[[in_usa(l) for l in
               by_cols['loc']]]
      loc  day  month  year  hi  lo
0  Dallas   28      3  2019  74  57
2    Nome   31      3  2019  31  26
```

5.5 Row Partition

Rather than mapping a `filter` function over multiple predicates, Pandas provides a `groupby` method for splitting up `DataFrames`. The value to use for each row's group selection can be a tuple containing some of that row's fields. As more general options, we can pass a function which computes the desired group identifier or a list of group identifiers to place each row in the corresponding group. Similar to the case in `#lang remora/dynamic`, the same comprehension we used for filtering can be used for partitioning as well.

```
>>> us__non_us =
    by_cols.groupby([in_usa(l) for l in
                            by_cols['loc']])
>>> us__non_us.get_group(True)
      loc  day  month  year  hi  lo
0  Dallas   28      3  2019  74  57
2    Nome   31      3  2019  31  26
>>> us__non_us.get_group(False)
      loc  day  month  year  hi  lo
1  Dublin    1      4  2019  11   5
3   Tunis   31      3  2019  21  12
```

5.6 Ergonomics

The most noticeable benefit to Pandas, over both other sections' representations of data frames as well as the list-of-dictionaries option, is the user-friendly output formatting. Annoyingly, using Standard ML records means repeating the column names in every table row, while using Racket structs means never showing column names. Standard ML also tends to silently reorder record fields according to its own purposes, which may cause related pieces of data to print quite far apart from each other. Sticklers about the term "REPL" might find Pandas output unsatisfying, as the printed result cannot be read back in as the object it represents. In principle, even that should be fixable by sprinkling delimiters in the right places.

The primary drawback of following a Pandas-like design for records in a rank-polymorphic language is its reliance on a large amount of *ad hoc* machinery. Functionality designed for too specific a purpose is more difficult to adapt, even to closely related tasks. For example, `groupby` bakes in the assumption that the resulting groups are disjoint and exhaustive. It is awkward to generalize to tasks like selecting several incrementally growing subsets of the data. A rank-polymorphic language should already include a good range of general-purpose array-manipulating primitives which can be used on columnar tables in much the same way as on vectors, matrices, or higher-dimensional tensors.

In the vein of type-system friendliness, Python's string-to-anything map structure is not very easy to handle, but the ability to compute field names via string operations is probably unneeded flexibility. The ability to concatenate data frames both vertically and horizontally can help with gathering data from multiple sources, but Racket structs and Standard ML records both make merging heterogeneous operations difficult.

6 Synthesis: Manipulating Records with Lenses

Now it is time to come up with a design for record data which accounts for the advantages and disadvantages discussed in the preceding sections.

The most important design goal for integrating record types with rank polymorphism is for construction and projection to be first-class functions. The use of special syntax rather than a constructor function means that a Standard ML programmer must manually define such functions in order to get lifting over tables. Python partially sidesteps the problem by emphasizing comprehension syntax, but even this becomes awkward when lifting over several arguments with differing frame shapes. Racket's take on this is far more notationally heavyweight due to having to declare a `struct` type in order to produce its constructor and projection functions. We propose instead a notation for record constructors. In s-expression syntax, a record constructor is written as

(`record` *fname*$_1$ `...` *fname*$_n$)

The value of a `record` form is an n-ary function on scalars. When applied to scalar arguments *arg*$_1$ through *arg*$_n$, it produces a record equivalent to Standard ML's

{*fname*$_1$ `=` *arg*$_1$, `...`, *fname*$_n$ `=` *arg*$_n$}

or Python's

{'*fname*$_1$' `:` *arg*$_1$, `...`, '*fname*$_n$' `:` *arg*$_n$}

Similar notation can even be offered as syntactic sugar with

{(*fname*$_1$ *arg*$_1$) `...` (*fname*$_n$ *arg*$_n$)}

expanding to

((`record` *fname*$_1$ `...` *fname*$_n$) *arg*$_1$ `...` *arg*$_n$)

We also need corresponding notation for projection functions, and ideally update functions. Lenses, arising from work by Foster *et al* [1] and popularized by libraries such as Haskell's `Control.Lens` [7], give a composable way to access elements of larger structures. Whether used individually or composed, they are ultimately consumed by functions corresponding to what to do at the field in question:

- (`view L`) produces a function which extracts the value of the field in its argument associated with the lens `L`
- (`set L`) produces a function which changes the field's value to a passed-in value, constructing a new record
- (`over L`) produces a function which updates the field's value according to a passed-in unary function, also constructing a new record

Most importantly for our purposes, calling `view`, `set`, or `over` on a record-field lens produces a record-consuming function, which we can then lift over structures of records. This also requires some notation for constructing a lens from a field name; we will use the syntax (`lens` *fname*). While `lens` is a keyword rather than a function, its result is a function.

We also propose syntactic sugar for constructing lens-based operations:

- `#_(`*fname* `...)` expands to
  `(view (compose (lens` *fname*`) ...))`
- `#=(`*fname* `...)` expands to
  `(set (compose (lens` *fname*`) ...))`
- `#^(`*fname* `...)` expands to
  `(over (compose (lens` *fname*`) ...))`

Preserving field order, as opposed to considering a record as an unordered collection of labeled values, allows Pandas to give much more readable output than Standard ML. Records with the same types of field values in different orders are isomorphic, but the difference between them is observable by a user within our intended mode of use. So we conclude that the proper design choice here is to consider records with different field orders non-equal.

Unlike previous sections, this is a design sketch rather than an explication of an existing system. It is not backed up by an implementation at this time, so the code presented below is not executable.

6.1 Table Creation

```
> (define dallas-temp
    {(loc "Dallas") (day 28) (month 3)
     (year 2019) (hi 74) (lo 57)})
> (#_(year) dallas-temp)
2019
> ([#_(hi) #_(lo)] dallas-temp)
[74 57]
```

Row-based construction of a table works as before.

```
> (define temp-readings
    [dallas-temp
     {(loc "Dublin") (day 1) (month 4)
      (year 2019) (hi 11) (lo 5)}
     {(loc "Nome") (day 31) (month 3)
      (year 2019) (hi 31) (lo 26)}
     {(loc "Tunis") (day 31) (month 3)
      (year 2019) (hi 21) (lo 12)}])
```

Since the "record literal" syntax is actually sugar for function application, it can lift over aggregate arguments with compatible frames, effectively assembling a data frame from its columns.

```
> (define temp-readings
    {(loc ["Dallas" "Dublin" "Nome" "Tunis"])
     (day [28 1 31 31])
     (month [3 4 3 3])
     (year 2019)
     (hi [74 11 31 21])
     (lo [57 5 26 12])})
```

6.2 Column Extraction

With rank polymorphism, a lens operation lifts from operating on a single record's field to operating on a column in a data frame—*i.e.*, the corresponding field in every row.

```
> (#_(loc) temp-readings)
["Dallas" "Dublin" "Nome" "Tunis"]
```

```
> {(loc   (#_(loc)   temp-readings))
   (day   (#_(day)   temp-readings))
   (month (#_(month) temp-readings))
   (hi    (#_(hi)    temp-readings))}
[{(loc "Dallas") (day 28) (month 3) (hi 74)}
 {(loc "Dublin") (day 1) (month 4) (hi 11)}
 {(loc "Nome") (day 31) (month 3) (hi 31)}
 {(loc "Tunis") (day 31) (month 3) (hi 21)}]
```

6.3 Column Update

As with view, the operations set and over generalize from a single record's field to a data frame's column. Because field updates are functions, they are easily composed.

```
> (define (normalize-temps (w 0))
    (define new-temp
      (select (in-usa? (#_(loc) w)) f->c id))
    ((compose (#^(lo) new-temp)
              (#^(hi) new-temp))
     w))
> (normalize-temps temp-readings)
[{(loc "Dallas") (day 28) (month 3) (year 2019)
 (hi 23.33) (lo 13.89)}
 {(loc "Dublin") (day 1) (month 4) (year 2019)
 (hi 11) (lo 5)}
 {(loc "Nome") (day 31) (month 3) (year 2019)
 (hi -0.56) (lo -3.33)}
 {(loc "Tunis") (day 31) (month 3) (year 2019)
 (hi 21) (lo 12)}]
```

6.4 Row Filter

We keep the same filter function from Section 3. Defining a boolean mask differs only slightly. The weather-loc field accessor we used when working with Racket structs can be replaced directly with a lens operation #_(loc).

```
> ((compose in-usa? #_(loc)) temp-readings)
[#t #f #t #f]
```

6.5 Row Partition

As in Section 3, we construct and then filter with a frame of several boolean masks. The same ways of building arrays of masks work in this setting as well, with struct field accessors replaced by view operations.

6.6 Ergonomics

Record construction syntax allows the scalar-level flexibility we had in Standard ML and Python: accessing a field depends only on the field name, with no requirement to define a type for the particular collection of names. We also gain the advantage we had with Racket structs by treating record notation as syntactic sugar for application of a particular function. That function lifts to higher-rank arguments, such as table columns. This avoids the need to explicitly declare a record-construction function for each collection of fields.

Using lenses to access data within records also avoids the need for Racket's type-specific field accessor functions. However, eventual efforts to introduce static types must allow row polymorphism or run into the same limitations as Standard ML, where field-manipulating code is tied at the type level to a particular collection of field names.

Tables with hierarchically nested columns can also be implemented in all three systems examined above, using a purpose-specific data structure in Pandas or nested structs and records for Racket and Standard ML respectively. However, lenses make record nesting much more palatable because a lens for a deeply nested subfield is constructible using function composition—this is much more lightweight notation than, for example, chaining field update functions for Racket-style structs. Point-free programming is the traditionally favored style in rank-polymorphic programming languages, so the use of composable lenses integrates well.

Beyond the Interaction of Orthogonal Features While we have been treating heterogeneous record data and rank polymorphism as independent, orthogonal language features, there may be something to gain from tighter coupling. For a fairly mild example, printing a single record requires including all of its field names, but when printing a large array of records, repeating the field names at each array element clutters the user's view with redundant information.

A farther-reaching possibility would be allowing records to appear in function position, with the record structure forming part of the frame shape. The individual field-cells' results would then be assembled into a record of results. This would allow the column-subset example to be written as something like

```
> (~(0){#_(loc) #_(day) #_(month) #_(hi)}
   temp-readings)
```

7 Other Related Work

In our exploration of how rank polymorphism interacts with heterogeneous records, we have chosen three languages' notions of record data with the goal of seeing different sets of advantages and disadvantages. Other rank-polymorphic languages have also taken their own approaches to heterogeneous data.

Futhark [3] is an array-oriented language for GPU programming, which includes record types [4]. Record access is a special syntactic form rather than a function, using the same notation for referring to both a field within a record and a name exported by a module.

R [10] is a domain-specific language for statistical analysis, with a `data.frame` class as a core piece of the language. This class and its operations served as the inspiration for Pandas and several other libraries in a variety of languages.

Gibbons demonstrated an embedding of rank polymorphism in Haskell via an `Applicative` type class instance [2]. However, Haskell's record system is historically a sore point in the language's design, lacking first-class record types and even prohibiting types with overlapping field names from coexisting in the same module.

Iverson's languages APL [5] and J [6] do not include record types, and this omission is a significant problem in the design of the language. However, heterogeneous arrays have some support. APL allows heterogeneous arrays directly, whereas J requires elements of a heterogeneous array to be boxed. Either option allows an alist-style implementation of records, but heterogeneous data is not the typical programming style in these languages.

8 Conclusion

We have presented a design for producing and consuming record data aimed at flexibility in scalar computation. Scalar-level flexibility then leads to easy data frame use when we combine records with rank polymorphism. This investigation serves as a useful guide for eventual extension of Remora's design to include records. Offering a small collection of mutually composable constructs—in our case, rank polymorphism, records, and conventional array-manipulation operations—avoids the need to build and export a large collection of task-specific tools.

References

[1] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 17 (May 2007). https://doi.org/10.1145/1232420.1232424

[2] Jeremy Gibbons. 2016. APLicative Programming with Naperian Functors (Extended Abstract). In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2976022.2976023

[3] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language*. Ph.D. Dissertation. University of Copenhagen, Universitetsparken 5, 2100 København.

[4] Troels Henriksen. 2017. Dot Notation for Records. https://futhark-lang.org/blog/2017-11-11-dot-notation-for-records.html

[5] Kenneth E. Iverson. 1962. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA.

[6] Jsoftware, Inc. [n. d.]. The J programming language. http://www.jsoftware.com/

[7] Edward Kmett. [n. d.]. lens: Lenses, Folds and Traversals. http://hackage.haskell.org/package/lens

[8] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The definition of standard ML: revised*. MIT press.

[10] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[11] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *European Symposium on Programming Languages and Systems*. Springer, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

[12] Satish Thatte. 1991. A type system for implicit scaling. *Sci. Comput. Program.* 17, 1-3 (Dec. 1991), 217–245. https://doi.org/10.1016/0167-6423(91)90040-5

[13] Mitchell Wand. 1991. Type inference for record concatenation and multiple inheritance. *Information and Computation* 93, 1 (1991), 1–15.