ShapeRank: Rank Polymorphism meets Reactive Streams

Gilad Bracha
F5 Networks/Shape Security
g.bracha@f5.com

Abstract

ShapeRank is a statically typed, functional programming language designed for machine learning, data analytics and reactive programming. All ShapeRank values are multidimensional streams, known as *hyperstreams*, and all operations are automatically lifted to process hyperstreams in parallel. This lifting originates in the APL [12] language family, and is known as *rank polymorphism*. ShapeRank extends rank polymorphism to hyperstreams. ShapeRank functions may be either *synchronous* or *reactive*. We introduce ShapeRank with a focus on its reactive behavior.

1 Introduction

Array programming originated in the early 1960s with Iverson's Turing-award winning APL [12]. Dataflow programming has a long, varied, and separate history. We bring these two threads together in a new programming language called ShapeRank. The ShapeRank programming language can be seen as extending APL style rank polymorphism to streams, or as extending dataflow programming with rank polymorphism.

In APL and its descendants, arrays have a *shape*, which is a vector of integer dimensions. The length of the shape is known as the *rank* of the array. One can write functions that operate polymorphically over arguments of different rank (hence the term rank polymorphism). In the simplest case, this implies simply mapping functions over arrays pointwise. For example, given two arrays a and b, the operation $a + b$ works for arrays of any rank: a and b can be scalars (which are considered arrays of rank 0), or vectors, or matrices etc. More complex rules allow for expressions such as $[1, 2, 3] + 2$, where the scalar 2 is replicated to form a 3-vector $[2, 2, 2]$ that can then be added pointwise to $[1, 2, 3]$. The precise rules vary somewhat between rank-polymorphic array programming languages. However, there is usually a requirement that all arrays are hyper-rectangles: Every element of an array must have the same shape, recursively.

The intuition behind an extension from arrays to streams is obvious: again, it is rooted in the idea of pointwise mapping of functions, but to streams instead of arrays. For example, given a stream of integers s, we might expect that $2 * s$ yields a stream whose elements are the elements of s, doubled.

However, there are some interesting implications:

1. A stream can have substreams of unknown size. We are no longer restricted to dealing with the regular hypercuboids of array programming, nor can we rely on that regularity.
2. For functions of multiple parameters, one immediately has to consider how to align the different arguments, which are streams of data that may arrive at varying rates.

In this paper, we mainly focus on the implications of item 2 above.

We might expect that given streams s_1 and s_2, the expression $s_1 + s_2$ denotes a stream consisting of the sums of the corresponding elements of s_1 and s_2. Even such a simple case is not as obvious as one might think. We have quietly decided that the + operator will synchronize its arguments (this seems implicit in the idea of *corresponding elements*). Yet this is not always what one wants.

Suppose we are interested in knowing when the sum of two independent inputs exceeds some threshold. Here we do not want to synchronize the two input streams. Rather, we want to update the sum whenever either input changes, that is, when either stream has new data. [1]

ShapeRank supports both scenarios by letting programmers choose whether to synchronize the input streams, or to react whenever data arrives on any one of them. Specifically, one can define a function as *synchronous* or *reactive*. A synchronous function will be invoked whenever *all* its inputs have new data; a reactive one, when *any* of the inputs has new data. The reactive behavior is the default.

ShapeRank is statically typed. Statically typechecking rank polymorphism is a topic of ongoing research [8, 9, 16], and ShapeRank has a unique approach to this problem, but that lies beyond the scope of this paper.

By design, ShapeRank is not Turing-complete. There are no looping constructs, and recursion is disallowed. All iteration is driven implicitly by the shape of the data. As a result, a program written in ShapeRank can never diverge. [2]

Below we explore the design of ShapeRank, focusing on its support for reactive programming. Section 2 gives an overview of the language. A detailed example follows in section 3 . We review the project's current status in section 4. Section 5 discusses related work, and section 6 concludes.

[1] Obviously, the domain of UI is another source of many examples where this kind of reactive behavior is preferred.

[2] Unless it is passed a divergent external function

2 Language Overview

2.1 Streams and Hyperstreams

Streams and hyperstreams are central to the language semantics.

A *stream* is a vector of unknown length. The unknown length is written ?, and signifies a potential infinity. Streams may nest, and so ShapeRank supports multidimensional streams. Ordinary vectors are regarded as finite streams; a vector of length n is a stream that happens to terminate after n elements.

All values in ShapeRank are *hyperstreams*. A hyperstream is either a scalar, or a stream whose elements are hyperstreams. A hyperstream has zero or more *dimensions*. The number of dimensions of a hyperstream is known as its *rank*. All elements of a hyperstream must have the same rank. A scalar hyperstream has zero dimensions, and therefore has rank 0; otherwise, the hyperstream has $1 + k$ dimensions, where k is the rank of of its elements. Each dimension has a size, which may be known or unknown. If the size is unknown, it is indicated by ?; otherwise it is a positive integer.

The *shape* of a hyperstream is a vector of its dimensions. All elements of a hyperstream must conform to the same shape. The shape of a scalar is the empty vector, which we write as []. If a hyperstream is not a scalar, its shape consists of the stream's size, followed by the shape of its elements. The rank of a hyperstream always equals the length of its shape.

Apart from the fact that the size of a stream may be unknown and unbounded, these definitions are essentially the same as for arrays in the APL language family.

2.2 Rank Polymorphism

How does the implicit lifting of functions to hyperstreams of various rank work? In this section we discuss the rules of rank polymorphism. The full rules are available in [4] and are closely related to those of languages such as J [1] and Remora [16], except for the extensions to streams. The goal here is to provide the intuition behind the rules for those unfamiliar with the details of array programming languages, with an emphasis on differences due to the extension to hyperstreams.

In the simplest case, rank polymorphism means that operations are applied pointwise to the scalar elements of hyperstreams. This implicit lifting of functions to hyperstreams is obvious for unary functions, e.g., $-[1, 2, 3]$ evaluates to $[-1, -2, -3]$. Essentially, every call $f(x)$ would be replaced by $map(f, x)$. In reality, things are more involved.

We break the process into four parts: framing, replication, zipping and mapping. The overall process is quite standard, following the model of the rank-polymorphic array programming languages mentioned above. Where ShapeRank differs is in the exact rules for replication (section 2.2.2) and zipping (section 2.2.3).

2.2.1 Framing

Suppose we want to sum the rows of a matrix. We might write this as: $sum([[2, 3, 4], [5, 6, 7]])$. We want *sum* to operate on vectors (or more generally, on one-dimensional streams) and we'll want to map *sum* on to each row of the matrix, not onto its individual elements.

We view the argument $[[2, 3, 4], [5, 6, 7]]$ as a *frame* consisting of a pair of 3-vectors (its rows). The rows are called the *cells* of the frame. We seek to map *sum* pointwise over the cells of the frame.

A frame is like a hyperstream, except that its base elements are not scalars but cells. A hyperstream may be framed in various ways. Our example matrix can be viewed as a pair of two 3-vectors as we suggested. Alternately, we can view it as a matrix, in which case the frame is the entire hyperstream, and the cells are scalars. A third way is to view the hyperstream as a frame of rank 0, which consists of a single cell of rank 2. In general, a hyperstream of rank n can be decomposed into frames and cells in $n + 1$ ways: a rank n hyperstream with rank 0 elements, a rank $n - 1$ hyperstream with rank 1 elements, ..., a rank 1 hyperstream with rank $n - 1$ cells, or a rank 0 frame with a single rank n cell.

When framing an argument x, each cell must itself be a hyperstream of rank $rank(p)$, the rank expected by the function's corresponding formal parameter p. For this reason, we require that formal parameters be annotated with an expected rank (see section 2.3 below). The rank of x can be larger than $rank(p)$. The difference, $\Delta = rank(x) - rank(p)$ determines the rank of the argument's frame. The shape of the frame is a prefix, of length P, of the shape of x. The shape of each cell is the suffix, of length $rank(p)$, of the shape of x. These rules completely determine the shape of the frame of an argument and of the frame's cells.

In our example, we have $rank(p) = 1, rank(x) = 2, \Delta = 1$ yielding a frame of rank 1, with cells of rank 1. The shape of the frame is a length $P = 1$ prefix of $[2, 3]$, the shape of x, namely $[2]$. The shape of the cells is $[3]$, determined by the length $rank(p) = 1$ suffix of the shape of x. And so *sum* will be invoked twice, each time on a distinct row of the input.

We generalize the above definitions to multiple arguments in the obvious way: for a function f of arity k, let p_i be the ith formal parameter of f and let $rank(p_i)$ be the declared rank of p_i. Given a call $f(x_1, \ldots, x_k)$, each argument x_i has a corresponding frame F_i. The cells of F_i must be hyperstreams of the expected rank $rank(p_i)$ so that the function can process them. Each argument requires the function to be called on it $\Delta_i = rank(x_i) - rank(p_i)$ times so that all data in the argument is processed. Thus $rank(F_i) = \Delta_i$.

The concept of framing plays a key role in the overall process of calling rank polymorphic functions, as described in the following three subsections.

2.2.2 Replicating

In a function with multiple formal parameters, each parameter may have a different rank, and so may each of the actual arguments. Each argument will have its own frame, potentially with its own shape. If we are to use the frame's shape to determine how to map the function, we must ensure all frames have a common shape.

As an illustrative example, consider the expression $2 * [1, 2, 3]$. We expect this to produce $[2, 4, 6] = [2*1, 2*2, 2*3] = [2, 2, 2] * [1, 2, 3]$. In other words we should automatically replicate arguments as needed, so that they have a common shape. How do we decide how arguments are replicated? Our rules again are very close to the traditional rules of rank polymorphism, but dealing with streams introduces a small twist.

Overall, the function f must be invoked enough times to process the argument with the highest ranked frame, known as the *maximal frame*. The common shape must therefore have length $N = max(P_i), i \in 1..k$. Argument frames whose rank is less than N must be expanded. This is done by a process of *replication*. We replicate the cells of lower ranked frames so as to obtain extended frames whose shape is compatible with the maximal frame. A frame of shape $[\delta_1, \ldots, \delta_m]$ can be extended to any shape $[\delta_1, \ldots, \delta_n]$, where $n \geq m$. Given a maximal frame of shape $[d_1, \ldots, d_N]$ and another argument frame of shape $[d'_1, \ldots, d'_m]$, extension is possible as long as $d_i = d'_i, 1 \leq i \leq m$. In other words, $[d'_1, \ldots, d'_m]$ must be a prefix of $[d_1, \ldots, d_N]$. This is known as the *prefix rule* in array programming, where all the dimensions are finite integers.

In our case, a small adjustment must be made, because for streams, some dimensions may not be integers, but ?. The unknown dimension is considered compatible with any fixed size dimension. Thus, if any of the argument frames has ? as its mth dimension, the mth dimension of the common shape is ?. Otherwise, the mth dimension of the common shape is d_m, where d_m is the mth dimension of the maximal frame.

Each argument frame is then extended to the common shape. If any argument frame's shape is incompatible with the common shape (namely, one of its dimensions is incompatible with the corresponding dimension in the common shape), we have an error.

2.2.3 Zipping

To map functions over multiple arguments (ShapeRank does not support currying and/or user defined higher order functions) we assume functions take tuples.

We can think of evaluating $[10, 20, 30] + [100, 200, 300]$ as computing $map(+[[10, 100], [20, 200], [30, 300]])$. In other words $map(+, zip([10, 20, 30], [100, 200, 300])$. In light of the previous sections, we are in fact zipping the (possibly extended) frames of all the arguments into a single common frame whose cells are tuples of arguments.

The definition of zip we use differs with function arity, but more importantly, between synchronous and reactive functions.

For synchronous functions of arity k, we use the function zip_{sync_k}. Given hyperstreams h_i, each with n_i^t available elements at an arbitrary time t, $i \in 1..k, k > 0$, the function zip_{sync_k} produces a hyperstream which at time t is defined as

$h_{sync} \triangleq (h_{1_1}, \ldots, h_{k_1}), \ldots, (h_{1_n}, \ldots, h_{k_n})$

where $n = min(n_i^t, i \in 1..k$.

An important property of zip_{sync_k} is that it synchronizes its inputs.

For reactive functions of arity k, we use the function zip_{react_k}, which produces a hyperstream h_{react}, whose first element is $(h_{1_1}, \ldots, h_{k_1})$. When the number of available elements of any of h_i increases, an element $(h_{1_{n_1}}, \ldots, h_{k_{n_k}})$ may be appended to h_{react}. The precise decision of when to add new elements to h_{react} is up to the implementation, which should strive to add elements promptly.

The value of h_{react} is non-deterministic. The first value is produced when all of the h_i have an available value, just as with h_{sync}. From that point on, however, new values may be added whenever any one of the h_i has a new available value. No guarantees are made as to whether such a value will be added and when. It is quite possible that some of the h_i will have a new available value, while others do not. In such cases, for any h_i that doesn't have a new value, the last available value is reused. It is also possible that multiple new values may have become available on any or all of the h_i, in which case some values will be skipped.

The implementation of zip_{react_k} is a best-effort, subject to complex engineering trade-offs, much like garbage-collection.

In an extreme case, zip_{react_k} might produce a single value, but this would be an unacceptably poor implementation. Alternatively, zip_{react_k} could behave just like zip_{sync_k}, and this too would be unacceptable.

2.2.4 Mapping

Having framed the arguments, replicated them as needed to produce extended frames and zipped these into a common frame whose cells are argument vectors, we can map a function on to the common frame.

The exact *map* function depends on the rank of the common frame. Scalars require no mapping, so we use $map = map0D = id$. In general, for a rank R frame $map = mapRD$.

2.3 Programs

A program consists of a program header, followed by a series of declarations and ending with an expression that is the value of the program. Both the header and the declarations are optional.

Program headers give the program a name, and bind program parameters. These parameters may be input streams

for the program to process, or they may be modules or external functions. ShapeRank's module system is beyond the scope of this paper. We will see the use of a program header with input streams and external functions in section 3.

Declarations are either functions or let bindings, which introduce variables. Apart from let bindings, variables may be introduced by formal parameter declarations in functions or via `where` clauses.

Functions consist of a function header, giving the name of the function, and listing its formal parameters, followed by the function body, which is an expression. As an example:

```
func sum(x: []) = reduce(+, 0, x)
```

Formal parameters always indicate their expected rank, if any. A rank of $n > 0$ is indicated by a series of n pairs of matched brackets. A one dimensional stream would be indicated by [], a two-dimensional one by [][] etc. For rank 0 (scalars) the indication is omitted. In some cases, no particular rank is expected; this can be indicated using the symbol ?. The actual dimensions and element type of a parameter will be inferred.

Functions are reactive by default, but may be defined as synchronous by prefixing them with the `sync` keyword, e.g.:

```
sync func twice(x) = 2*x
```

Expressions are either literal hyperstreams, variable references, `where` clauses, or function calls (operators such as +, −, ∗ etc. are sugar for function calls). All expressions denote hyperstreams.

3 An Example

As an example of using ShapeRank for a reactive task, we will consider a simple build system. A build system can be viewed as a very simple dataflow application. In the most straightforward interpretation, files are read, and the data is fed into compilers, and the compilation results are fed into a linker to produce an executable.

More realistically, a build system would invoke tools like compilers and linkers as external functions. In this variant, the inputs are filenames, but they feed into nodes that monitor the files for changes, producing steams of notifications that are fed into other nodes that activate the compilers. The compilation nodes likewise produce notifications that drive linking. Obviously, we want nodes to fire when they get a notification on any one of their input streams.

Encoding this into ShapeRank, we'll assume the following external functions;

$getFile(filePath : String) : [?]String$

The *getFile* function is a wrapper for a file-watcher. Given a file path, it echoes it onto a stream - once when we verify the file exists, and then whenever the file changes.

$clang(path : String, deps : [?]String, flags : String) : String$

The function *clang* is expected to call the compiler with the name of a file to be compiled, a list naming any dependencies and a string of command-line flags. It returns the name of the object file resulting from the compilation.

$basicLink(target : String, deps : [?]String, flags : String) : String$

basicLink is similar to *clang*, but it invokes a linker. It expects the name of the binary output, a list of object file paths, and a flags string.

The program accepts these three functions as parameters; this is the mechanism we use to provide external functions to ShapeRank. Within the ShapeRank program, we define two functions, *cc* and and *link*. The *cc* function takes the same parameters as *clang*, but transforms them before passing them along. The first parameter is passed to *getFile*, producing a stream of strings. Even though *clang* is declared to take a single string as its first argument, the rules of rank polymorphism allow us to invoke it with a stream of strings instead. If *clang* had a single argument, the effect would be to invoke *clang* pointwise on every element of the stream. Since the stream has an element placed on it every time the named file changes, this would invoke the compiler on the file every time a change occurs.

There are multiple parameters however. How do things work? Let's first look at how we deal with the second parameter, *deps*. We first call *getFile* on *deps*; *getFile* takes a string, but will be mapped pointwise on to the list of dependencies. The result is a list of streams, one per dependency. Each stream consists of the dependency name, echoed every time the underlying dependency changes. This list of streams is fed into *zip*, which converts a list of streams into a stream of lists.

Zip is reactive, and so it will output a list onto the stream whenever any of the incoming streams has new data, reusing old data from others. Hence its output has a new element whenever any of the dependencies changes. Each element of the output list is taken from one of streams of the incoming list. The list therefore consists of the names of the various dependencies.

The combined effect of feeding the two streams into *clang* is to invoke it each time a new value appears on either stream, using the latest value from each input. The *flags* parameter is a scalar, and will be replicated on each call. And so, when all is said and done, the compiler is invoked whenever any file changes, be it the main file to be compiled or any of its dependencies. The result of *clang* is ordinarily a string, but it is of course replicated into a stream containing the results of all the invocations - in other words, the name of the object file is echoed every time it is updated.

The treatment of *link* and *basicLink* is essentially the same. Using these functions, we can write out the necessary steps to compile and link a build target in what would normally be a build script. However, the program behaves more like a make file. It only rebuilds those artifacts that actually need to be rebuilt. And since the program is running continuously, artifacts are rebuilt whenever any of their dependencies

changes, and only then. A mature ShapeRank implementation would also parallelize the build process. The complete program is shown below.

```
program build(
  getFile: (filePath: String): [?] String,
  clang: (path: String, deps: [?]String, flags: String): String,
   basicLink: (target: String, deps: [?]String, flags: String
): String)
func cc(path, deps: [], flags) =
  clang(getFile(target), zip(getFile(deps)), flags)
func link(
  target,
  deps: [][],
  flags
) =  basicLink(target, zip(deps), flags);
link("myApp", [
 cc(
 "Users/gbracha/shapeRank/f5/shapeRank/examples/a.c",
 ["Users/gbracha/shapeRank/f5/shapeRank/examples/b.h"],
  "-c"
 ),
 cc(
 "Users/gbracha/shapeRank/f5/shapeRank/examples/b.c",
 ["Users/gbracha/shapeRank/f5/shapeRank/examples/b.h",
  "Users/gbracha/shapeRank/f5/shapeRank/examples/c.h"],
   "-c"
 ),
 cc(
 "Users/gbracha/shapeRank/f5/shapeRank/examples/c.c",
 ["Users/gbracha/shapeRank/f5/shapeRank/examples/c.h"],
  "-c"
)], "")
```

4 Status and Future Work

The ShapeRank prototype is available at [17]. The prototype is intended as a reference implementation, to experiment with the language design. We intend to extend the design to better support querying over streams, possibly in the style of streaming SQL. Once the design is complete, we hope to build a performant implementation. We'd also like to have a formal semantics for ShapeRank.

5 Related Work

The two principle influences on ShapeRank are array programming [1, 3, 9, 10, 12, 15] and dataflow. In particular, Lucid [18] is a dataflow language designed around data streams. However, array programming does not support streams, and dataflow does not support rank polymorphism.

The assumption that arrays are hyperrectangular is central to the design and implementation of most array programming languages. The concept of explicit boxing is used to work around these restrictions. In contrast, ShapeRank relies on type inference to determine when streams are in fact of known finite dimensions and can be treated as arrays.

Functional reactive programming (FRP) [6, 7, 14] is centered around event streams and *behaviors*. Behaviors are continuous time varying values. ShapeRank streams can represent events, but there is no notion of continuous values.

Synchronous reactive programming [2, 5] is also obviously related, but differs in that it imposes a synchronous discipline on the entire program. A program in these languages can be seen as a synchronous function over the program's input streams. Within the program, there is no rank polymorphism.

In Python, rank polymorphism is implemented dynamically by the NumPy library [11] where it is referred to as *broadcasting*. Numpy is not intended to support streaming or reactive programming.

Dart's async* methods [13] return a stream when called, and then suspend until there is a demand for content from the stream. Then the body of the method runs until a yield statement, which evaluates an expression, places the resulting value on the stream that the method had returned, and suspends until stream content is asked for once again. The resulting streams are consumed by await-for loops; each iteration demands a stream value and suspends until it is available, processes the value and repeats indefinitely (modulo returns or breaks).

Unlike ShapeRank code, the generator function can loop indefinitely, intentionally or not. Our approach makes asynchrony implicit, avoiding the problem of contagion characteristic of async/await style constructs.

Frameworks such as Rx, Kafka etc. expose streams via APIs in mainstream languages, without language support.

6 Conclusions

We have presented the key elements of the design of ShapeRank, a language featuring rank polymorphism over multi-dimensional streams. ShapeRank allows functions to be defined reactively or synchronously. To our knowledge, the combination of rank polymorphism and stream processing has not been explored previously. ShapeRank supports both array programming and stream/dataflow programming in the same language, and allows for the concise description of reactive workflows, which we have illustrated via the example of software build processes.

References

[1] J programming language system/documentation. Available online at https://code.jsoftware.com/wiki/System/Documentation.

[2] Berry, G., and Gonthier, G. The Esterel synchronous programming language: Design, semantics, implementation, 1992.

[3] Borror, J. A. Q for mortals an introduction to q programming. Available online at https://code.kx.com/q4m3/.

[4] Bracha, G. ShapeRank language specification (draft). Available online at https://github.com/f5devcentral/shapeRank/tree/main/docs/spec.

[5] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: A declarative language for programming synchronous systems. In *In*

14th Symposium on Principles of Programming Languages (POPL'87). ACM (1987).

[6] Cooper, G. H., and Krishnamurthi, S. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems* (Berlin, Heidelberg, 2006), P. Sestoft, Ed., Springer Berlin Heidelberg, pp. 294–308.

[7] Elliott, C., and Hudak, P. Functional reactive animation. In *International Conference on Functional Programming* (1997).

[8] Gibbons, J. Aplicative programming with Naperian functors. In *European Symposium on Programming* (April 2017), H. Yang, Ed., vol. 10201 of *LNCS*, pp. 568–583.

[9] Grelck, C. Single assignment C (Sac): High productivity meets high performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary* (2012), V. Zsók, Z. Horváth, and R. Plasmeijer, Eds., vol. 7241 of *Lecture Notes in Computer Science*, Springer, pp. 207–278.

[10] Grelck, C., and Scholz, S.-B. Sac: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming 34*, 4 (2006), 383–427.

[11] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[12] Iverson, K. *A Programming Language*. Wiley, 1962.

[13] Meijer, E., Milliken, K., and Bracha, G. Spicing up Dart with side effects. *Queue 13*, 3 (Mar. 2015), 40:40–40:59.

[14] Meyerovich, L. A., Guha, A., Baskin, J. P., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages and Applications* (2009), S. Arora and G. T. Leavens, Eds., ACM, pp. 1–20.

[15] Scholz, S.-B., Herhut, S., Grelck, C., and Penczek, F. Single assignment C tutorial. Ppopp 2010, Bangalore, India. Tech. Rep. 498, School of Computer Science, University of Hertfordshire, 2010.

[16] Shivers, O., Slepak, J., and Manolios, P. Introduction to rank-polymorphic programming in Remora (draft).

[17] team, T. S. ShapeRank programming language github repository. Available online at https://github.com/f5devcentral/shapeRank.

[18] Wadge, W. W., and Ashcroft, E. A. *LUCID: The data flow programming language.*