An Introduction to Array Programming in Petalisp

Marco Heisig
Sandoghdar Division
Max Planck Institute for the Science of Light
Erlangen, Germany
marco.heisig@mpl.mpg.de

Abstract

Petalisp is a purely functional array programming language embedded into Common Lisp. It provides simple yet powerful mechanisms for reordering, broadcasting, and combining arrays, as well as an operator for element-wise mapping of arbitrary Common Lisp functions over any number of arrays.

This introduction covers the process of writing high-performance array programs in Petalisp and showcases its main concepts and interfaces. It continues with a simple example of an iterative method and some benchmarks, and concludes with a tour of the Petalisp implementation and a discussion how it achieves high performance and a low memory footprint.

ACM Reference Format:
Marco Heisig. 2024. An Introduction to Array Programming in Petalisp. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.11062314

1 Introduction

At the 11th European Lisp Symposium in Marbella we proposed a lazy, functional array programming language with significant potential for automatic parallelization. We showed a working prototype and some promising benchmarks that place its performance somewhere above NumPy and below C++[2]. Our hope was that this prototype could be quickly extended to cover more sophisticated problems, and to actually reach the performance of C++. This endeavor turned out to be substantially harder than expected. A key challenge we had to overcome was that of choosing memory layouts with good spatial and temporal locality, and to fairly distribute work across multiple cores. Now, after six years of hard work, we can finally say we have overcome these problems. We proudly present the first production-quality version of Petalisp, and are looking forward to receiving community feedback.

Petalisp is free software. The full source code and many examples can be found at https://github.com/marcoheisig/Petalisp. It can be installed with Quicklisp by typing `(ql:quickload :petalisp)`.

2 Related Work

Array programming is a discipline with a long history. The first array programming language was Kenneth E. Iverson's APL[6], whose terse notation and productivity benefits inspired a multitude of derivatives. Many recent array programming languages, e.g., Repa[7] or Futhark[5], have shifted towards the functional programming paradigm, but sacrificed some amount of interactivity and dynamism on the way. Petalisp delivers all the benefits of purely functional programming while retaining the interactive nature of Common Lisp. A project with a similar goal is the APL compiler April[8], that also targets Common Lisp.

ELS'24, March 06–07, 2024, Vienna, Austria

https://doi.org/10.5281/zenodo.11062314

3 Concepts

There are many approaches to designing a programming language. The one extreme is the *big ball of mud* approach, where more and more potentially competing features are added over time. The other extreme is to have a minimal set of orthogonal features. Petalisp pursues the latter extreme: It features only one data structure — the lazy array, six ways to reorder arrays, one function for combining arrays, and one function for mapping Common Lisp functions over any number of arrays.

3.1 Lazy Arrays

All data manipulated by Petalisp is represented as lazy arrays, which are similar to regular Common Lisp arrays except that their contents cannot be accessed directly, and that they support a more general notion of an array shape. Where the shape of a regular array is defined by its list of dimensions, the shape of a lazy array is defined by a list of ranges. Each range is a set of integers x defined by an inclusive lower bound a, an exclusive upper bound b, and a step size s in the following way:

$$\begin{aligned} a \le x < b \quad & (x \text{ is bounded by a and b}) \\ s|(x-a) \quad & (s \text{ divides } (x-a)) \\ a, b, s, x \ & \in \mathbb{Z} \end{aligned}$$

Formally, the shape of each lazy array is defined as the Cartesian product of its sequence of ranges. Informally, this means that lazy array have a numbering that doesn't necessarily start from zero, and that each axis can have holes in it as long as those holes are all regularly spaced.

Lazy array shapes have their own shorthand notation, which is a list consisting of tilde symbols and integers. Each tilde must be followed by one, two, or three integers, describing the size, start and end, or start, end, and step of the range in the corresponding axis, respectively. In this notation, a 2×3 array has a shape of `(~ 2 ~ 3)`, and a vector with step size two and four elements has the shape `(~ 0 7 2)`.

Lazy arrays can be created as copies of existing regular arrays or scalars with the `lazy-array` constructor. All Petalisp functions use this constructor to automatically convert their arguments to lazy arrays, so there is usually no need to call it explicitly. For

efficiency reasons there exists a second constructor called `lazy-index-components` for creating lazy arrays whose contents are fully described by a range and an axis. This second constructor is special because the resulting lazy arrays have a memory footprint of zero.

3.2 Evaluation

As mentioned in section 3.1, there is no way to access elements of a lazy array directly. Instead, a user has to convert lazy arrays into the equivalent regular arrays with an explicit function call. The main interface for doing so is the function `compute`. It receives any number of lazy arrays, moves them such that their shapes have a start of zero and a step size of one, and returns the equivalent regular arrays.

3.3 Lazy Map

There are only two mechanisms with which Petalisp communicates with its host language Common Lisp. The first mechanism is the conversion from Common Lisp arrays to Petalisp lazy arrays and vice versa. The second mechanism is that of mapping Common Lisp functions over lazy arrays to obtain new lazy arrays. The higher-order function for doing so is called `lazy` — a rare case of a function whose name is an adjective. The first argument to `lazy` must be a function f of k arguments, followed by k lazy arrays $a_0, \ldots, a_{k-1}$ that are broadcast to have the same shape. The result is a lazy array r of the same shape as the arguments, whose element at index I is defined as

$$r(I) := f\left(a_0(I), \ldots, a_{k-1}(I)\right).$$

Listing 1 illustrates the behavior of `lazy`. There is also another function for lazy mapping called `lazy-multiple-value` that can be used to map functions with multiple return values and gather each of those values in a separate lazy array.

```
(compute (lazy #'*))
 => 1

(compute (lazy #'+ 2 3))
 => 5

(compute (lazy #'+ 2 #(1 2 3 4 5)))
 => #(3 4 5 6 7)

(compute (lazy #'* #(2 3) #2A((1 2) (3 4))))
 => #2A((2 4) (9 12))
```

Listing 1: Examples for using the function `lazy`.

3.4 Lazy Reshape

True to the goal of being a minimalist programming language, Petalisp offers a single function, named `lazy-reshape`, for moving data. It can be used to select, reorder, or broadcast elements of a particular lazy array. All its operations can be described as the superposition of six elementary operations, which are shown in Figure 1.

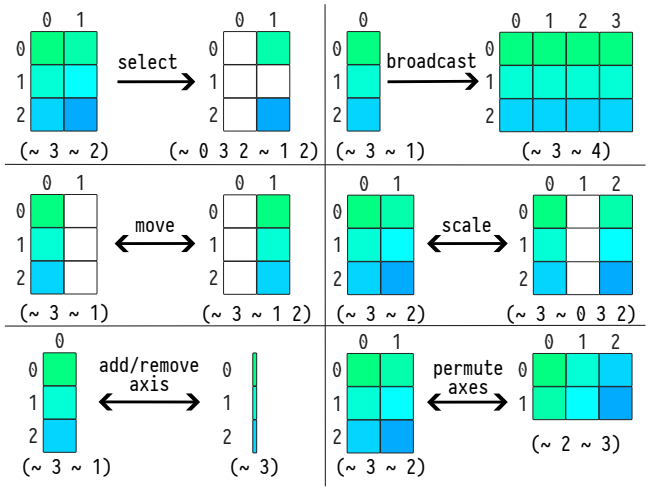

Figure 1: The six elementary reshape operations.

The first argument to lazy reshape is the lazy array that is being reshaped, and all the remaining arguments are so-called *modifiers* that are processed left-to-right and each describe a particular combination of elementary operations. One possible modifier is a shape, in which case the result is a lazy array of that shape and the modification is a combination of selecting, broadcasting, and moving of data. Another possible modifier is that of a transformation, which describes some combination of moving, scaling, permuting, adding, or removing of axes. Transformations can be created using a lambda-like syntax with the `transform` macro, or using the `make-transformation` constructor. Examples of the various kinds of modifiers and their effect are shown in Listing 2

```
(compute (lazy-reshape #(1 2 3 4) (~ 1 2)))
 => #(2)

(compute (lazy-reshape #(1 2 3 4) (~ 2 ~ 3)))
 => #2A((1 1 1) (2 2 2))

(compute (lazy-reshape #(1 2 3 4) (~ 4 ~ 2)))
 => #2A((1 1) (2 2) (3 3) (4 4))

(compute (lazy-reshape #2A((1 2) (3 4))
           (transform i j to j i)))
 => #2A((1 3) (2 4))

(compute (lazy-reshape #(1 2 3 4)
           (transform i to (- i))))
 => #(4 3 2 1)
```

Listing 2: Examples for using the function `lazy-reshape`.

3.5 Lazy Fuse

The final piece of functionality that makes up Petalisp is that of fusing multiple arrays into one. The function for doing so is called

`lazy-fuse`. It takes any number of non-overlapping lazy arrays, determines the shape that covers all these lazy arrays, and returns the array with that shape that contains all the data of the original arrays. An error is signaled in case any of the supplied lazy arrays overlap, or if they cannot be covered precisely with a single shape.

4 The Standard Library

4.1 Moving Data

Shapes and transformations aren't the only valid modifiers accepted by `lazy-reshape`. It also accepts modifiers that are functions that take a shape of the lazy array being mutated, and return any number of further modifiers as multiple values. These functions are called *reshapers*, and they are a generalization of NumPy's relative addressing with negative indices. Petalisp features three built-in functions for constructing reshapers: `peeler`, for removing some of the outer layers of a lazy array, `deflater`, for shifting a lazy array to have a start of zero and a step size of one, and `slicer`, for selecting a particular subset of a lazy array using relative indices.

4.2 Reducing

The function `lazy-reduce` combines the contents of k arrays with a function of $2k$ arguments and k return values. It is an improved version of the multiple value reduction we presented at the 12th European Lisp Symposium in Genova[3].

4.3 Sorting

The function `lazy-sort` constructs a sorting network that sorts the supplied lazy array along the first axis using some predicate and optional key.

4.4 Differentiating

The function `differentiator` can be applied to a list of lazy arrays and a list of gradients at those lazy arrays to return a function that computes the gradient of each input of any of those lazy arrays. This is achieved by using our type inference Typo to compute the derivatives of Common Lisp functions. This functionality can serve as the staring point for writing a machine learning toolkit in Petalisp.

5 Example: Jacobi's Method

Listing 3 shows an implementation of a simple numerical scheme in two dimensions. Although this code is purely functional and has a very high level of abstraction, our benchmark results in Figure 2 show that it has a multi-core performance that is close to hand-optimized C++ code, and even outperforms the popular machine learning framework JAX.

6 The Implementation

Each call to `compute` or any of the other evaluation functions entails a full run through our optimization and code generation pipeline. With a careful choice of algorithms, data structures, and caching schemes, we managed to squeeze the time to execute this entire pipeline to something on the order of a few hundred microseconds. Because of these extremely fast compilation times we can

```
(defun lazy-jacobi-2d (u)
 (with-lazy-arrays (u)
  (let ((p (lazy-reshape u (peeler 1 1))))
   (lazy-overwrite-and-harmonize u
    (lazy #'* 1/4
     (lazy #'+
      (lazy-reshape u (transform i j to (1+ i) j) p)
      (lazy-reshape u (transform i j to (1- i) j) p)
      (lazy-reshape u (transform i j to i (1+ j)) p)
      (lazy-reshape u (transform i j to i (1- j)) p)))))))
```

Listing 3: Jacobi's method on arrays of rank two.

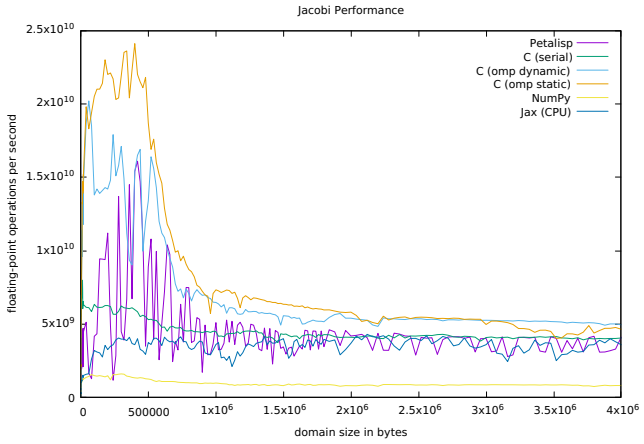

Figure 2: Benchmark results for various implementations of Jacobi's method.

pretend that Petalisp is an interactive language, yet receive all the performance advantages of static compilation.

6.1 Data flow graphs

Initially, each Petalisp program is represented as a data flow graph whose nodes are lazy arrays and whose edges are direct data dependencies. This graph is assembled by invoking Petalisp functions such as `lazy-reshape` or `lazy-fuse`. Several optimizations are already carried out during graph assembly: consecutive reshape operations are combined into one, nodes with no effect are discarded, and fusions of reshape operations that are equivalent to a single broadcasting reshape are represented as such.

The most important optimization at this stage is to narrow down the element types of all lazy arrays produced by lazy mapping, which is a prerequisite for choosing a memory-efficient representation during execution. To do so, we wrote a portable and extremely fast type inference library named Typo that is available at https://github.com/marcoheisig/Typo. Typo can derive the (approximate) return types of almost all standard Common Lisp functions, and it can rewrite calls to polymorphic functions with specialized arguments into calls to more specialized functions.

6.2 Kernels and Buffers

Once a data flow graph of lazy arrays is submitted for evaluation, it is converted to the Petalisp intermediate representation, which is a bipartite graph of kernels and buffers. Each kernel represents some number of nested loops whose body contains loads, stores, and function calls. Each buffer represents a virtual memory region. We developed an algorithm that ensures that most values are produced and consumed in the same kernel so that the size and number of necessary buffers is minimal.

Once a program is converted to this intermediate representation, it is subject to several optimizations: kernels and buffers are rotated in a way that maximizes memory locality, all shapes and iteration spaces are normalized to have a starting index of zero and a step size of one, and buffers that are involved in reduction-like patterns are eliminated and replaced by additional instructions in the adjacent kernels.

6.3 Partitioning

The next step in the optimization pipeline is to break up buffers and the kernels writing to them into shards of roughly equal computational cost, which is a prerequisite for scheduling them onto parallel hardware. We developed an iterative partitioning algorithm that minimizes the amount of synchronization and communication.

6.4 Scheduling

The partitioned intermediate representation is fed into our scheduling algorithm, which is a variant of Blelloch's parallel depth-first scheduler algorithm[1] with several tweaks that improve memory locality. Our custom scheduler has several advantages over general purpose schedulers: It has full knowledge about the origins of each load and the users of each store and the partitioning step has already ensured that all tasks have roughly the same size.

6.5 Allocation

At the end of the scheduling phase, each buffer shard is assigned a particular memory allocation in the following way: buffer shards of similar size are all grouped into one bin, and within each worker and each particular bin, a register-coloring algorithm is used to assign an allocation to each buffer shard while keeping the total number of allocations small.

6.6 Code Generation

When executing the schedule, each kernel is converted to an optimized Lisp function that is invoked on three arguments: The iteration space of a kernel shard, the memory corresponding to each buffer shard, and a vector of all functions that are called in the kernel. Because kernel compilation is rather costly, each kernel is first converted to a hash-consed minimal representation that can be used as a key for caching, and compilation only occurs when a kernel is invoked for the first time.

We already have code generators for turning kernels into Common Lisp code and for turning a subset of kernels into C++ or CUDA code. Right now, our strategy is to use C++ and GCC when possible, and Common Lisp code otherwise. In the future, we plan to make the C++ generator obsolete by using SIMD optimized Common Lisp instead. Doing so would build on our previous work on `sb-simd` that we presented at the 15th European Lisp Symposium in Porto[4].

7 Conclusions

We presented a data flow programming language that masquerades as a Common Lisp library for manipulating arrays. The language stands out by having an extremely simple set of core operations, a versatile standard library, and a mature implementation. A major achievement of our implementation is that it can already outperform optimized C++ code in certain cases — both in execution time and memory consumption. In the past, the manipulation of high-dimensional arrays in Common Lisp has been tedious and often inefficient. Petalisp addresses this issue thoroughly, and turns Common Lisp into an excellent tool for all sorts of massively parallel array programming tasks.

References

[1] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, mar 1999. ISSN 0004-5411. doi: 10.1145/301970.301974. URL https://doi.org/10.1145/301970.301974.

[2] Marco Heisig. Petalisp: A common lisp library for data parallel programming. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, ELS2018. European Lisp Scientific Activities Association, 2018. ISBN 9782955747421.

[3] Marco Heisig. Lazy, parallel multiple value reductions in Common Lisp. In *Proceedings of the 12th European Lisp Symposium*, European Lisp Symposium, 2019. ISBN 978-2-9557474-3-8. doi: 10.5281/zenodo.2642164. URL https://european-lisp-symposium.org/static/proceedings/2019.pdf.

[4] Marco Heisig and Harald Köstler. Closing the performance gap between lisp and c. In *Proceedings of the 15th European Lisp Symposium*, ELS2022. Zenodo, March 2022. doi: 10.5281/zenodo.6335627. URL https://doi.org/10.5281/zenodo.6335627.

[5] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. Universitetsparken 5, 2100 København, 11 2017.

[6] Kenneth E Iverson. *A Programming Language*. John Wiley & Sons, Nashville, TN, December 1962.

[7] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding parallel array fusion with indexed types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 25–36, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315746. doi: 10.1145/2364506.2364511. URL https://doi.org/10.1145/2364506.2364511.

[8] Andrew Sengul. April: Apl compiling to common lisp. In *Proceedings of the 15th European Lisp Symposium*, European Lisp Symposium. Zenodo, March 2022. doi: 10.5281/zenodo.6381963. URL https://doi.org/10.5281/zenodo.6381963.