Lazy, parallel multiple value reductions in Common Lisp

Marco Heisig
FAU Erlangen-Nürnberg
Cauerstraße 11
Erlangen 91058, Germany
marco.heisig@fau.de

ABSTRACT

Reductions, folds, or catamorphisms are an important component of every functional programmer's toolbox. However, common manifestations of these operators can only operate on a single sequence at once and don't have any potential for parallel execution.

We present a new, parallelizable reduction operator that can simultaneously reduce k arrays at once, using a function with $2k$ arguments and k return values. We then discuss an efficient implementation of this new reduction operator as part of the Petalisp project.

CCS CONCEPTS

•**Software and its engineering →Functional languages; Data flow languages; Parallel programming languages;** *Just-in-time compilers;*

KEYWORDS

Common Lisp, Reductions, Lazy Evaluation, Parallelism

ACM Reference format:
Marco Heisig. 2019. Lazy, parallel multiple value reductions in Common Lisp. In *Proceedings of the 12th European Lisp Symposium, Genova, Italy, April 01–02 2019 (ELS'19),* 6 pages.
DOI: 10.5281/zenodo.2642164

1 INTRODUCTION

The reduction is one of the most versatile tools of the functional programmer. Figure 1 defines the exemplary reduction operator `fold`. Despite its simplicity, it captures the essence of what we find in the standard libraries of Scheme, Haskell, Common Lisp and many other programming languages: recursive processing of a data structure and combination of values with a binary function.

```
(defun fold (f z l)
  (if (null l)
      z
      (fold f (funcall f (first l) z) (rest l))))
```

Figure 1: A simple reduction operator: `fold`.

The simple four line function in figure 1 is quite powerful, as long as we confine ourselves to the domain of lists. Depending on the supplied binary function and initial value, we can express a variety of concepts:

- **sum**
  `(fold #'+ 0 numbers)`
- **product**
  `(fold #'* 1 numbers)`
- **maximum**
  `(fold #'max 0 non-negative-numbers)`
- **reversal**
  `(fold #'cons '() list)`
- **filtering**
  ```
  (fold (lambda (i j) (if (oddp i) (cons i j) j))
        '() list)
  ```

The good news is that due to its tail-recursive structure, our `fold` function can be run very efficiently on a serial processor. The bad news, however, is that this function is completely unsuited for execution on parallel hardware. This gets apparent if we visualize the data flow of a particular call, as in figure 2.

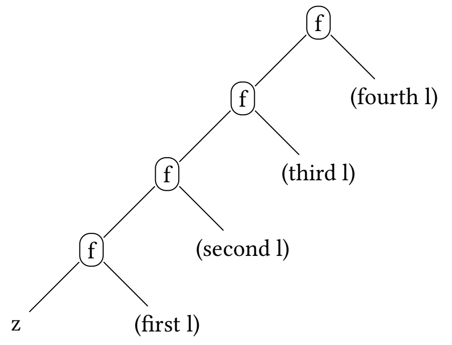

Figure 2: Data-flow graph of a call to fold.

A call to `fold` results in a dependency chain that has the same length as the list being worked on and, consequentially, there is zero potential for parallel execution. In a world of ubiquitous multicore processors, this is a huge and growing problem. Or, as Guy Steele put it, "foldl and foldr Considered Slightly Harmful"[9].

In this paper, we present a reduction operator that has inherent parallelism, addresses simultaneous reduction of multiple sequences and reductions of multi-dimensional arrays.

2 PETALISP

This work has been developed as part of the Petalisp project[5, 6]. The goal of Petalisp is to provide a model for data parallel programming that fits nicely into the existing general purpose programming language Common Lisp. Petalisp programs are composed of only a tiny number of core operators — parallel map, parallel reduce, affine-linear data motion and data fusion — and only a single data structure — the lazy array. Lazy arrays can be evaluated, i.e., turned into Lisp arrays with the function `compute`.

As a convenience feature, Petalisp functions implicitly convert arguments that are regular Lisp arrays to lazy arrays, and converts all other arguments to lazy arrays of rank zero. Because of this implicit conversion, most of the discussion in this paper can be carried out without worrying about Petalisp at all, apart from inserting an occasional call to `compute` for explicit evaluation.

The minimalist set of features in Petalisp unlocks a lot of powerful optimizations, but means that its programs are fundamentally limited by the semantics of its core operators. That is one of the reasons why we put so much effort into the definition of parallel reduction.

3 RELATED WORK

We do not explicitly list each and every kind of reduction throughout the computer science landscape, but report only operators that have either inherent parallelism, or special support for handling multiple values.

- The Scheme languages includes a variety of operators for folding and reducing list elements in SRFI-1[8]. Many of them can operate on multiple lists at once, in which case the combining function will receive one argument for the accumulating value and one argument per list. However, no special provisions exist for reducing multiple values at once, and none of the functions is inherently parallel.
- The parallel programming language NESL[1] permits hierarchical reduction similar to ours, since its language core supports nested parallel constructs. However, the advantage of our technique is that it is embedded into the powerful general purpose language Common Lisp instead of being a standalone tool. Furthermore, we are not aware that NESL has capabilities for handling multiple values.
- The MPI standard[4] for distributed programming includes primitives for reducing potentially distributed data in parallel. It also permits reductions with user-defined functions and, with the right annotations, changes to the order of evaluation. But all MPI reductions are limited to reducing a single array with a binary function.
- The programming model of MapReduce[2] is similar to the model provided by Petalisp. Here, processes are split into a *Mapper* function to express embarrassingly parallel tasks and a *Reducer* function describe how data is to be accumulated. While MapReduce has excellent support for parallelism, each *Reducer* function takes only a single key-value pair and therefore suffers from the same limitations as most other reductions.
- Connection Machine Lisp[10] features a syntactic construct for parallel, unordered reduction, named β. This construct has not only influenced the design of our reduction operator, it is also the origin of our operator's name.

4 OUR TECHNIQUE

What follows is the definition of our reduction operator β.

function **β** *f* *array* &rest *more-arrays* → *result**

The supplied function f must accept $2k$ arguments and return k values, where k is the number of supplied arrays in *array* and *more-arrays*. All supplied arrays must have the same shape S, which is the cartesian product of some ranges, i.e., $S = r_1 \times \ldots \times r_n$, where each range r_k is a set of integers, e.g., $\{0, 1, \ldots, m\}$. Then β returns k arrays of shape $s = r_2 \times \ldots \times r_n$, whose elements are a combination of the elements along the first axis of each array according to the following rules:

(1) If the given arrays are empty, signal an error.
(2) If the first axis of each given array contains exactly one element, drop that axis and return arrays with the same content, but with shape s.
(3) If the first axis of each given array contains more than one element, partition the indices of this axis into a lower half l and an upper half u, such that $r_1 = l \cup u$ and such that either $|l| = |u|$, or $|l| = |u| + 1$. Then split each given array into a part with shape $l \times s$ and a part with shape $u \times s$. Recursively process the lower and the upper halves of each array independently to obtain $2k$ new arrays of shape s. Finally, combine these $2k$ arrays element-wise with f to obtain k new arrays with all values returned by f. Return these arrays.

4.1 A Simple Example

Let us illustrate this definition with a simple example. If we apply β to a binary function f and a vector with four elements, we start out with $k = 1$ and the shape $S = \{(0), (1), (2), (3)\}$. We are dealing with a rank one array, so the result will be a rank zero array.

Since the given array is neither empty, nor has just a single element, we start out with an application of rule 3. That means we split the given array into a lower half with indices $\{(0), (1)\}$ and an upper half with indices $\{(2), (3)\}$ and process each half recursively. The lower half is again subject to rule 3 and split into a part with the sole index $\{(0)\}$ and one with the sole index $\{(1)\}$. Further recursive processing of each of these one-element arrays results in an application of rule 2, where the given rank one array are turned into equivalent rank zero array. These rank zero arrays are returned to the previous application of rule 3, where their sole elements are combined with the function f to form the content of the rank zero array that is the value of the lower half. The upper half is processed analogously. Finally, these arrays are combined by yet another application of the function f to obtain the final result. Figure 3 illustrates this process.

4.2 Discussion

We will now motivate and justify the individual design decisions we made when defining β.

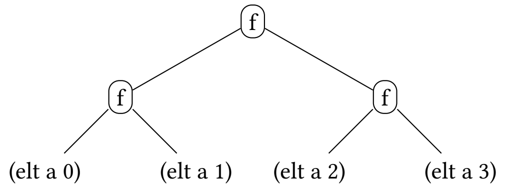

Figure 3: Data-flow graph of a call to β on a four element vector.

Handling of Empty Arrays. It is common for reduction operators to take an explicit initial element that is returned when processing an empty sequence. The function β, however, simply signals an error for this case. This behavior is best explained by looking at the two cases how initial elements are typically used:

In the first case, the initial element has the same type as the elements of the sequence and both arguments of the combining function therefore have the same type. In this case, it is sometimes possible to find a suitable initial element, e.g., zero for addition. But in general there is no such element. A prominent example for this general case is computing the maximum of a sequence of integers. The only sane initial element would be $-\infty$, which is not an integer.

In the second case, the initial element has a different type than the elements of the sequence and, consequently, the arguments of the combining function must be heterogeneous. In essence, the initial element acts as an accumulator that is threaded through all sequence elements sequentially. This would directly conflict with our goal to allow parallel execution. Luckily, there is a suitable equivalent for accumulation in our β operator, which is to convert the array of values to an array of accumulators first, and to define the combining function such that it merges given accumulators[1]. The difference can be seen in figure 4, where we use lists as accumulators and the function append as combining function. The function α that we use in this figure is the lazy, parallel mapping operator of Petalisp. Its semantics is similar to that of `cl:map`.

```
(defparameter *a* #(1 2 3 4 5 6))

;; inherently serial
(reduce #'cons *a* :initial-value '() :from-end t)

;; parallel alternative
(compute (β #'append  (α #'list *a*)))
```

Figure 4: Two ways to convert an array to a list.

The Name β. Some readers might frown at the decision to use a greek letter as a function name. One frequent complaint is that this renders any code using this function non-portable. The other frequent complaint is that modern keyboards offer no easy way to insert Greek letters. To the first complaint, we reply that parallel programming is already outside of the scope of the Common Lisp specification, so the concern would only apply to the hypothetical implementations that support concurrency, but not Unicode. To the second complaint, we reply that there is plenty of IDE support for inserting nonstandard characters, and that parallel programming is so hard that the time spent typing should be negligible in contrast to the time spent thinking.

[1]Some readers might worry about the cost of creating an array of accumulators first. But thanks to lazy evaluation, Petalisp can eliminate this temporary array and move the creation of the accumulator directly into the reduction.

Rank Zero Arrays Instead of Scalars. By definition, or function β can never return scalar values, only arrays of rank zero. This could be an annoyance for the simple, frequent case of reducing vectors. The solution we developed here is that Petalisp treats scalars and arrays of rank zero interchangeably. And, most importantly, when calling `compute` to trigger explicit evaluation, all arrays of rank zero are automatically converted to scalars.

Choice of Axis. According to our definition, reductions apply only to the first axis of a given array. Another option would have been to reduce along a specified axis and to reduce the entire array if no axis is explicitly specified. While this would add some convenience for the user, it would make β less orthogonal to the existing set of Petalisp primitives. The choice of axis can already be achieved by permuting an array's indices, which is already supported by another Petalisp primitive. And the reduction of an entire array with rank n can be emulated by n successive reductions. Figure 5 illustrates these techniques.

Subdivision Strategy. The current subdivision strategy is to split the first axis of an array into two equal halves, until the axis has been reduced to a single index. As it can be seen in figure 3, the effect is that all array elements are effectively combined along the nodes of a binary tree. One might wonder whether a different or more flexible subdivision strategy could be more efficient, but we decided not to pursue this thought further until we have an excellent implementation of reduction on a binary tree.

One argument in favor of this reduction order is that it produces deterministic results. Since all Petalisp programs are just a combination of core operators, and all other core operators are already deterministic, we gain the property that all Petalisp programs are fully deterministic — a very desirable property for a parallel programming language.

Arrays Only. Our definition of β requires that all its arguments but the combining function are arrays (or, lazy arrays) and not, e.g., arbitrary sequences. The reason for this is that efficient execution requires that both the shape and the elements of each argument must be accessible in $O(1)$ time. However, it is conceivable that future versions of Petalisp will also support lazy arrays whose backing storage is a user defined sequence with fast random access, e.g., as proposed by Rhodes[7].

Multiple Values. Coming from statically typed functional languages, one might wonder why we bother with functions returning multiple values, when we just could have used tuples and let the compiler optimize them away. One reason is that this way, even code without sufficient static type information can be run efficiently and without additional consing. The other reason is that we would have had to introduce static, immutable tuples into Common Lisp and force every user of our reduction operator to use them, whereas multiple values are already part of the language.

5 EXAMPLES

In the following section, we show how our reduction operator can be used in practice.

5.1 Numeric Accumulation

In this first example, in figure 5, we show how β can be used to express the sum and product of some numbers. This illustrates also how some seeming deficiencies like lack of an initial value can be overcome with a suitable abstraction. In this case, we call this abstraction β*. It correctly handles the case of receiving an empty array, and, for further convenience, reduces the whole array if no explicit axis is supplied.

```
(defun β* (f z x &optional axis)
  (cond ((empty-array-p x)
         z)
        ((integerp axis)
         (β f (exchange-axes x 0 axis)))
        ((loop until (zerop (rank x))
               do (setf x (β f x))
               finally (return x)))))

(defun sum (x &optional axis)
  (β* #'+ 0 x axis))

(defun product (x &optional axis)
  (β* #'* 1 x axis))
```

Figure 5: Using β for numeric accumulation.

5.2 Computing the Maximum and its Index

In this second example, in figure 6, we show how to perform a reduction on multiple values. Our goal is to efficiently obtain both the maximum of a vector, and the corresponding index. To do so, the function `max*`, supplies two arrays to β — the array itself, and an array of the same shape containing the indices of the axis zero corresponding to each array element. These two arrays are then reduced with a four argument function that forwards either the two left arguments or the two right arguments, depending on which side yields the larger value.

As discussed later in section 6, the function `max*` is probably more efficient than a programmer would assume by looking at its definition. Not only can large parts of it be run in parallel, it is also possible to completely eliminate the lazy array that is given as the last argument to β, by computing its elements into a function of the currently processed index.

6 IMPLEMENTATION

In figure 7, we show a naïve implementation of β. The only *user-visible* difference between this naïve code and the one Petalisp actually uses is that there is no implicit broadcasting of argument arrays, no error handling, and no support for lazy arrays. But what this code shows is that a naïve implementation will always be prohibitively slow. Neither the dimensions of the given arrays, nor their rank, nor their element type are known at compile time. Not even the number of arrays k is known statically. To tackle this problem regardless, we have to make use of higher-order functions and relatively expensive constructs like `multiple-value-list` and `(apply #'aref)`.

```
(defun max* (x)
  (β (lambda (lv li rv ri)
       (if (> lv rv)
           (values lv li)
           (values rv ri)))
     x (indices x 0)))
```

Figure 6: Computing the maximum element and its index.

We see that the crucial question regarding an efficient implementation of our proposed β operator is how to deal with the large amount of compile time uncertainty. One possible approach would be to write or generate multiple versions of the code for each possible invocation, e.g., using the technique of Strandh[3]. The problem is that in our case, the space of possible arguments and types is extremely large. Assuming we wanted to create special versions just for the case of reducing up to three arrays with a rank below 3 and for every specialized array element type. Then, assuming an implementation with 20 specialized array types[2], we would end up with $2(20^1 + 20^2 + 20^3) = 16840$ different versions. Such an amount of specialization is unreasonable, even on a modern computer with plenty of memory.

What we do instead is that we generate specialized reduction functions on demand, using the existing framework of Petalisp. This has multiple advantages:

- Data flow analysis prevents unnecessary evaluation. If parts of the result of a reduction are not used, the corresponding inputs will also not be computed.
- If the inputs of a reduction are lazy arrays, the code that computes the contents of these arrays can often be inlined directly into the reduction, thus avoiding an unnecessary intermediate array.
- Specialized code is cached efficiently, such that future invocations with a similar signature can reuse the previously compiled code.
- The programmable type inference engine of Petalisp can often statically deduce the element type of the results of a reduction and allocate them in a suitable specialized array.
- The size of the argument arrays is known during code generation. This makes it possible to generate and use different variants, e.g., to avoid thread parallelization when the workload is known to be small.

[2]Having 20 specialized array types is a conservative estimate. SBCL on a 64bit architecture recognizes 34 subtypes of `array`, CCL even 38.

```
(defun β (f array &rest more-arrays)
 (let* ((arrays (list* array more-arrays))
        (k (length arrays))
        (r (array-dimension array 0))
        (dims (rest (array-dimensions array)))
        (results
          (loop repeat k
                collect (make-array dims))))
  (map-indices
   (lambda (indices)
    (mapcar
     (lambda (o v)
      (setf (apply #'aref o indices) v))
     results
     (multiple-value-list
      (labels
         ((divide-and-conquer (start end)
            (if (= start end)
                (values-list
                 (mapcar
                  (lambda (a)
                    (apply #'aref a
                           (list* end indices)))
                  arrays))
              (multiple-value-bind (ls le us ue)
                    (split-range start end)
                  (values-list
                   (subseq
                    (multiple-value-list
                     (multiple-value-call f
                       (divide-and-conquer ls le)
                      (divide-and-conquer us ue)))
                    0 k))))))
        (divide-and-conquer 0 (1- r))))))
   dims)
  (values-list results)))

(defun split-range (start end)
  (let ((mid (floor (+ start end) 2)))
    (values start mid (1+ mid) end)))

(defun map-indices (fn dims)
  (if (null dims)
      (funcall fn '())
      (apply #'alexandria:map-product
             (alexandria:compose fn #'list)
             (mapcar #'alexandria:iota dims))))
```

Figure 7: A possible implementation of β.

In order to support the new reductions, we also had to make several changes to the Petalisp internals. The most challenging task was to add support for functions returning multiple values. This change has profound implications, as it turns what used to be data-flow trees into directed acyclic data-flow graphs, and because it touches many optimization passes, such as common subexpression elimination and hoisting of loop invariant code.

We are happy to report that this transition is now complete, and that Petalisp now has full support for functions returning multiple values. These changes affect not only reductions, but also parallel mapping. It is now also possible to, e.g., map the function `floor` over a single array to obtain one array with all the quotients and one array with all the remainders.

To give a glimpse into the workings of our code generator, we show in figure 8 a part of the code generated during the first evaluation of a call to the function `max*` from figure 6 on a vector. This snipped of generated code shows how the number of values has been turned into a compile time constant, how the reference to the second array has been reduced to `(identity index)` and how the reference to the first array has been lowered to a call to `row-major-aref` with a simple offset.

```
...
(labels
  ((divide-and-conquer (min max)
     (declare (type fixnum min max))
     (if (= min max)
         (let ((index (+ min (* #:g3 #:g4))))
           (let* ((v (row-major-aref a0 index))
                  (i (identity index)))
             (values v i)))
        (let ((mid (+ min (floor (- max min) 2))))
           (multiple-value-call
               (lambda (l0 l1 r0 r1)
                 (multiple-value-bind (r0 r1)
                     (funcall f l0 l1 r0 r1)
                   (values r0 r1)))
             (divide-and-conquer min mid)
            (divide-and-conquer (1+ mid) max))))))
 (divide-and-conquer 0 (/ (- #:g5 #:g3) #:g4)))
...
```

Figure 8: An excerpt from the code generated for a call to `max*`.

7 PERFORMANCE

We have not yet implemented all optimizations that we envision, so it is too early for a detailed performance analysis. But we can already outline the most important performance characteristics of our technique. Our measurements show that for large vectors, our operator is about half as fast as SBCL'S built-in function `cl:reduce`, despite being vastly more general. The downside of on-demand code selection or generation is that it incurs a constant overhead of several microseconds, making it unsuitable for reductions of small arrays. However, we expect that we can lower this constant overhead in the future by switching to more efficient data structures and by caching some expensive intermediary steps.

8 CONCLUSIONS AND FUTURE WORK

We have presented a reduction operator β that is simple, powerful and has plenty of inherent parallelism. Most importantly, our reduction operator supports accumulation of multiple values at once. Thus, it greatly extends the range of programs that can be expressed as a single reduction. e.g., for finding both the minimum and maximum of a sequence, or for accurate summation of floating point numbers, using a second value to accumulate errors.

We have carefully presented our design considerations, especially with respect to inherent parallelism and simplicity. The value of simplicity is still underappreciated in modern parallel programming. We think that in order to obtain both correctness and speed, it is sometimes better to go for clean, robust approaches — such as reducing along a binary tree only — instead of chasing after the last few percent of performance.

As a second contribution, we have presented an implementation technique — on-demand compilation of specialized code at run time — that allows us to turn this operator into efficient, highly specialized code. To do so, we use the existing data-flow analysis and compiler infrastructure of Petalisp. All our code is freely available under a copyleft license[3].

This paper marks the end of the design process of the parallel programming library Petalisp. This doesn't mean we are finished with Petalisp development, but we will now focus exclusively on under-the-hood improvements, such as better thread-level parallelization, faster dispatch, SIMD vectorization and, ultimately, distributed parallelization.

9 ACKNOWLEDGMENTS

We would like to thank all the people in the `#petalisp` IRC channel.

REFERENCES

[1] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[3] Irène Durand and Robert Strandh. Fast, maintainable, and portable sequence functions. In *Proceedings of the 10th European Lisp Symposium*, ELS2017. European Lisp Scientific Activities Association, 2017.

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015. URL https://www.mpi-forum.org/docs/.

[5] Marco Heisig. Petalisp: A common lisp library for data parallel programming. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, ELS2018, pages 1:4–1:11. European Lisp Scientific Activities Association, 2018. ISBN 978-2-9557474-2-1.

[6] Marco Heisig and Harald Köstler. Petalisp: Run time code generation for operations on strided arrays. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2018, pages 11–17, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5852-1. doi: 10.1145/3219753.3219755. URL http://doi.acm.org/10.1145/3219753.3219755.

[7] Christophe Rhodes. User-extensible sequences in common lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622138. URL http://doi.acm.org/10.1145/1622123.1622138.

[8] Olin Shivers. Srfi-1: List library, 1998. URL https://srfi.schemers.org/srfi-1/srfi-1.html.

[9] Guy L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. *SIGPLAN Not.*, 44(9):1–2, August 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596551. URL http://doi.acm.org/10.1145/1631687.1596551.

[10] Guy L. Steele, Jr. and W. Daniel Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 279–297, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319870. URL http://doi.acm.org/10.1145/319838.319870.

[3]https://github.com/marcoheisig/Petalisp