Lazy, Parallel Multiple Value Reductions in Common Lisp

Marco Heisig
Chair for System Simulation, FAU Erlangen-Nürnberg
01.04.2019

Before we begin . . .

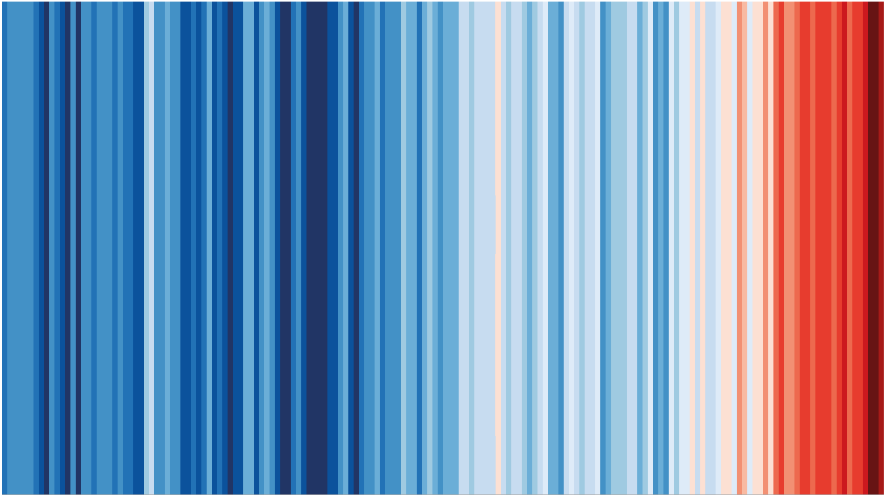


Table of contents

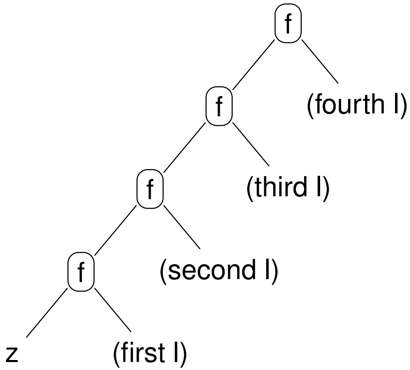
Motivation

Reductions are Awesome!

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

- **sum**
  `(fold #'+ 0 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)
  ```

Problem #1 - Parallelism



- Long serial chain of dependencies.
- Execution time will always be time(f) $\cdot$ length(l).
- Exascale computers expected in 2021.

“foldl and foldr Considered Slightly Harmful” – Guy Steele

Problem #2 - Multiple values

- **Life, as it should be:**

```
(reduce #'fn values :initial-value iv)
```

- **Life, as it is:**

```
(loop for value in values
      for elt across aux
      for idx from 0
      for acc-1 = (fn-1 value elt idx)
      for acc-2 = (fn-2 acc-1 elt idx)
      finally (return (values acc-1 acc-2)))
```

Goal: Reductions on multiple streams of data at once.

The Goals

- **Parallelism**
  $O(\texttt{log}(N))$ runtime on a sufficiently parallel machine.
- **Multiple values**
  Gather multiple quantities from multiple sources.
- **Laziness**
  Programmers should not have to cripple their source code to avoid allocation of intermediate data.
- **Array Programming**
  Support for multi-dimensional arrays.
- **Performance**
  Competitive to a good `cl:reduce`.

Context: The Petalisp Project

- A Common Lisp library for elegant parallel programming.
- The core data structures are lazy, strided arrays.
- All operations are deterministic and purely functional.
- Petalisp has only four core operators. Parallel reduction is one of them.
- Arrays are evaluated by calling `compute`.

Interested?

```
(ql:quickload :petalisp)
```

`https://github.com/marcoheisig/Petalisp`

`/join #petalisp`

The Function β

Definition

(defun β (f array &rest more-arrays) . . .)

- f must accept $2k$ arguments and return k values, where k is the number of supplied arrays.
- All supplied arrays must have the same shape $S = r_1 \times \ldots \times r_n$, where each range r_k is a set of integers, $\{0, 1, \ldots, m\}$.
- 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.

It remains to clarify how we combine elements of the first axis.

The Reduction Rules

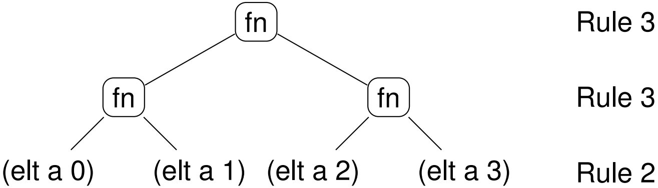
- k arrays of dimension n and shape S
- f is a function from $2k$ arguments to k values
- $n - 1$ dimensional output shape s

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 the resulting k arrays of shape s.
3. Otherwise
   - Split each array into a lower and an upper half.
   - Recurse into each of the two halves.
   - Combine the $2k$ resulting arrays of shape s element-wise with f.
   - Return the resulting k arrays of shape s.

A Simple Example

Example: `(β #'fn (vector a b c d))`

- The number of arrays k is 1.
- The input shape S is $(\{0, 1, 2, 3\})$.
- The output shape s is (), i.e. the result is a scalar.



Parallelism

	`(reduce #'+ array)`	`(β #'+ array)`
Number of Additions	$N - 1$	$N - 1$
Dependency Tree Depth	$N - 1$	$\lceil \log_2(N) \rceil$

$\Rightarrow$ The function β is well suited for parallel computing.

Multiple Values

Computing both the maximum element and its index:

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

Look Ma, no loops!

```
(multiple-value-call #'compute (max* #(2 4 6 1 3)))
  → 6
  → 2
```

Multiple Values and Multiple Dimensions

Computing both the maximum element and its index:

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

...works for multi-dimensional arrays, too!

```
(m-v-c #'compute (max* #2A((2 4) (6 1))))
  → #(6 4)
  → #(1 0)
```

Implementation

Implementing β is Hard

The function β has many degrees of freedom:

- The number of supplied arrays *k*.
- The rank of the supplied arrays *d*.
- The element type of each supplied array.

And this is without taking lazy evaluation into account!

Our reference implementation is terribly slow, with gems like

```
(values-list
  (subseq
    (multiple-value-list
      (multiple-value-call f
        (divide-and-conquer ls le)
        (divide-and-conquer us ue)))
    0 k))
```

Making β Fast

- For classical sequence functions, it is common to define multiple specialized versions.
- We cannot use this trick, because we'd require DE^k versions, where D is the supported number of dimensions and E is the number of specialized array element types.

What we do instead:

- Compute a normalized problem description.
- Turn this problem description into efficient Lisp code.
- Use `cl:compile` to generate a fast evaluator.
- Invoke the compiled function on the supplied arrays.
- Cache the compiled function, using the normalized problem description as key.

Result: `(β #'+ v)` can actually inline `#'+`!

The Petalisp JIT-Compiler

Each Petalisp evaluation consists of the following steps:

1. Broadcasting, Type Inference, Shape Checking
2. Data-flow Optimization
3. IR-Conversion
4. Normalization
5. Scheduling
6. Code Generation
7. Compilation
8. Execution

- Thanks to memoization, the steps 6. and 7. can usually be skipped.
- The steps 5. and 8. can usually overlap.
- The challenge is getting the steps 1. to 4. fast enough.
- We are down to a few microseconds, but need to get better.

Optimization Showcase - a call to max*

```
(labels ((divide-and-conquer (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))))))
 ...)
```

Future Challenges

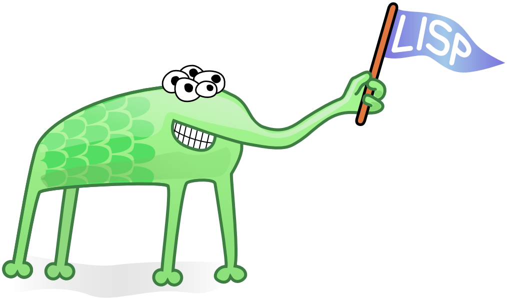
Challenges for the next months:

- Reduce the latency of `compute`.
- Add proper multi-threading.
- Further tweak the generated code.

Challenges for the next few years:

- Distributed Computing
- GPU offloading

Thank you!



Questions or remarks?