Bulk Operations on Indexed Collections

Jan-Willem Maessen
IFIP WGPL, 1 Mar 2012

My Background

- Clovers (Lynn Stein et al.):
  Inheritance using delegation, using CLOS MOP.
- Id (Nikhil & Arvind):
  - Implicitly-parallel functional programming language
  - Hindley-Milner, algebraic types, ML-ish syntax
- pH (parallel Haskell):
  - Id with Haskell syntax & type system.
- Making trouble about Java Memory Model
- Eager Haskell
  - Haskell via resource-bounded eager evaluation
- Fortress
  - Implicit parallelism, immutability
- Now: help Make The Web Faster

My Biases

- Libraries central to programmer's day-to-day experience of a language
  - Java collection classes
  - C++ STL
  - JavaScript DOM interface
  - Design language to enable tasteful libraries

- Really interested in parallelism
  - Including lock-free and wait-free algorithms

- I'm a functional programmer at heart

Summary: Big Idea

- Summations and list constructors and loops are alike!

$$\sum_{i \leftarrow 1:1000000} x_i^2$$

$$\langle\, x_i^2 \mid i \leftarrow 1:1000000 \,\rangle$$

```
for i ← 1:1000000 do
  x_i := x_i^2
end
```

  > Generate an abstract collection
  > The *body* computes a function of each item
  > Combine the results (or just synchronize)
  > In other words: map and reduce
- Whether to be sequential or parallel is a separable question
  > That's why they are especially good abstractions!
  > Make the decision on the fly, to use available resources

Indexed Collections

Arrays

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"

Finite Maps

"Sun"
'i'

"Trog"
'n'

"Slab"
'h'

"Tub"	"Peg"
'e'	's'

Sets

"Sun"	"Dock"	"Trog"

"Git"

"Tub"

"Fin"

Indexing Collections with Collections

```
inconsistent :: Region -> Puzzle -> Bool
inconsistent roi p =
    any isEmpty [ p[c] | c <- elements roi]

inconsistent :: Region -> Puzzle -> Bool
inconsistent roi p =
    any isEmpty p[roi]
```

What I think I want

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"

1		3	4				8		10

"Sun"		"Trog"	"Brick"				"Tub"		"Fin"

- Preserve structure of indexing collection
  - Here, indexing an array with a set yields a set
- Question: what to do when the elements of the collection indexed don't support creation of something like the indexed collection?
  - Example: no equality predicate on array elements, can't create a set.

What may be more sensible

Consider sets to have identical keys and values
Preserve the key space of indexing collection

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"

1	3	4	8	10
1	3	4	8	10

1	3	4	8	10
"Sun"	"Trog"	"Brick"	"Tub"	"Fin"

Seems to work for maps as well

"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"
3.1	41.5	9.2	6.53	5.8	9.7	9.3	2.38	4.6	2.6

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"

1	2	3	4	5	6	7	8	9	10
3.1	41.5	9.2	6.53	5.8	9.7	9.3	2.38	4.6	2.6

Questions

- Treatment of absent indices? Array vs set?
- Type of the indexing operation?
    - Not everything is a collection
    - Natural index type, and collections of indices
- Who is responsible for implementation?
    - Is this just a map over the indexing collection?
    - Beat O(m log n) when indexed collection is a tree
- Should we materialize the collection at all?

```
inconsistent :: Region -> Puzzle -> Bool
inconsistent roi p =
    any isEmpty p[roi]
```

Another example

```
removeSingletons :: Region -> Puzzle -> (Region, Puzzle)
removeSingletons roi p = (roi', p')
  where singletons =
            set [ c | c <- elements roi, size (p!c) == 1 ]
        elims =
            accumArray union empty puzzleBounds
                [ co | c <- elements singletons,
                       v <- elements (p!c),
                       co <- crossOuts c v ]
        p' = arrayZipWith difference p elims
        roi' = unions [ regions!c | c <- elements
singletons]
```

Combining corresponding indices

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"

1	2	3	4	5	6	7	8	9	10
3.1	41.5	9.2	6.53	5.8	9.7	9.3	2.38	4.6	2.6

1	2	3	4	5	6	7	8	9	10
"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"
3.1	41.5	9.2	6.53	5.8	9.7	9.3	2.38	4.6	2.6

And again on maps

"Sun"	"Dock"	"Trog"	"Brick"	"Slab"	"Git"	"Limb"	"Tub"	"Peg"	"Fin"
3.1	41.5	9.2	6.53	5.8	9.7	9.3	2.38	4.6	2.6

"Sun"
'i'

"Trog"
'n'

"Slab"
'h'

"Tub"	"Peg"
'e'	's'

"Sun"
3.1
'i'

"Trog"
9.2
'n'

"Slab"
5.8
'h'

"Tub"	"Peg"
2.38	4.6
'e'	's'

Questions

- Notation?
  - zip / zipWith are kind of terrible
  - Join operator? What about zipWith?
  - This is potentially an n-ary operation
- Return type?
  - Both args dense / both args sparse obvious...
  - First arg dense, second sparse?
  - First arg sparse, second dense?
  - Different kinds of sparse map (hash vs tree)
- Who's driving the operation?
  - Tricky again in non-uniform case
  - Keep the asymptotic complexity low & predictable
- Should we materialize?

How I got into this

- Treatment of zip on Fortress collections

- Database-style join operations
  - With predictable preformance
  - With notation that reflects operational behavior

- Array languages
  - Ability to index arrays with ranges:

```
a[2:17,3:19:2]
```

Preliminary Decisions

- Put the rightmost collection in charge
- Permit specialization of operations
  - Multimethod dispatch helps a lot here
- Don't materialize the results