It’s the data, stupid

Ryan R. Newton

2015-08-30
2002, Martin Rinard
You functional programmers have it completely wrong. You’re obsessed with code. “Beautiful code” — but it’s all about the data.

(paraphrased)
Which Map implementation is better optimised for memory?  

Submitted 9 days ago by alexeyraga

I am stream-processing lots of data and accumulating some stuff from the stream along the road. This data I accumulate is simply a map (or, sometimes, a map of maps) where a ByteString is a key and Int is a value. Consider a word counting example, it is somehow close to what I am doing. The data itself isn't that huge, it only takes ~200Mb when written on disk, but it takes ~15-18 gigabytes in memory when I use Data.Map. I have tried things like IntMap and HashMap, doesn't help much in terms of memory. Judy arrays aren super fine, but since they can only accept Word as keys they are not very useful in my situation.

So here is the question: is there anything "better" implementation of a Map that is more optimised for memory consumption? The current ratio 200MB:18GB doesn't seem to be very usable...
Which Map implementation is better optimised for memory?  

Submitted 9 days ago by alexeyraga

I am stream-processing lots of data and accumulating some stuff from the stream along the road. This data I accumulate is simply a map (or, sometimes, a map of maps) where a ByteString is a key and Int is a value. Consider a word counting example, it is somehow close to what I am doing. The data itself isn't that huge, it only takes ~200Mb when written on disk, but it takes ~15-18 gigabytes in memory when I use Data.Map. I have tried things like IntMap and HashMap, doesn't help much in terms of memory. Judy arrays aren super fine, but since they can only accept Word as keys they are not very useful in my situation.

So here is the question: is there anything "better" implementation of a Map that is more optimised for memory consumption? The current ratio 200MB:18GB doesn't seem to be very usable...
Hmm, we do focus a lot on the code

- Code
  - INLINABLE, SPECIALIZE
  - Core-to-core simplifier, optimizations

- Data
  - Poor Show/Read story compared to other langs
  - Sometimes bloated memory layout
    - Can’t unpack elements in “[Int]”
What about *data-structure* transformations & optimizations?

- Many old ideas
- Where are they in 2015?
What about data-structure transformations & optimizations?

- Many old ideas
- Where are they in 2015?

Unrolling Lists*

Zhong Shao†
Princeton University

John H. Reppy‡
AT&T Bell Laboratories

Andrew W. Appel†
Princeton University

Abstract

Lists are ubiquitous in functional programs, thus supporting lists efficiently is a major concern to compiler writers for functional languages. Lists are normally represented as linked cons cells, with each cons cell containing a car (the data) and a cdr (the link); this is inefficient in the use of space, because 50% of the storage is used for links. Loops and recursions on lists are slow on modern machines because

dependent on the previous one. With modern superscalar hardware, these dependences are a serious bottleneck.

In order to save on storage for links, “cdr-coding” was proposed in the 1970’s [15, 13, 8, 9, 6, 5]. Its main idea is to try to avoid some links by arranging for the second cons cell to directly follow the car of the first, and to encode that information in several bits contained in the car field of the first cell; thus the first cell does not need a cdr field at all.
What about *data-structure* transformations & optimizations?

- Many old ideas
- Where are they in 2015?

---

**21 years old!**

**Unrolling Lists**

Zhong Shao†
Princeton University

John H. Reppy‡
AT&T Bell Laboratories

Andrew W. Appel†
Princeton University

---

**Abstract**

Lists are ubiquitous in functional programs, thus supporting lists efficiently is a major concern to compiler writers for functional languages. Lists are normally represented as linked cons cells, with each cons cell containing a *car* (the data) and a *cdr* (the link); this is inefficient in the use of space, because 50% of the storage is used for links. Loops and recursion on lists are slow on modern machines because dependent on the previous one. With modern superscalar hardware, these dependences are a serious bottleneck.

In order to save on storage for links, "cdr-coding" was proposed in the 1970's [15, 13, 8, 9, 6, 5]. Its main idea is to try to avoid some links by arranging for the second cons cell to directly follow the car of the first, and to encode that information in several bits contained in the car field of the first cell; thus the first cell does not need a cdr field at all.
What about data transformation?

- Many old ideas
- Where are they 21 years old!

Unrolling

Zhong Shao
Princeton University

John H. Reif
AT&T Bell Laboratories

Abstract

Lists are ubiquitous in functional programs, thus supporting lists efficiently is a major concern to compiler writers for functional languages. Lists are normally represented as linked cons cells, with each cons cell containing a car (the data) and a cdr (the link); this is inefficient in the use of space, because 50% of the storage is used for links. Loops and operations on lists are slow on modern machines because...
Generic list length
Generic list sum
Specialized list length
Specialized list sum
The low level decisions affect all programs and matter the most for performance.
Specialize/monomorphize all datatypes: the right default?

- DSLs:
  - WaveScript
  - Ur/Web
- MLton
- Mu
- Hugs experiment (Mark Jones)
- C++ templates
Request for collaboration

- We’re working on:
  - GC workarounds & Serialization formats (CompactNF)
  - Concurrent data structures & msg passing
    - Talk to Omer Agacan, Ryan Scott, & I

- Ed Kmett & Ryan Yates are working on primops for unboxed mutable structs
  - space efficient mutable data structs

- Other projects?