Standard library data structures for Spiral

Jan Špaček

April 2016

1 Introduction

Spiral [12] is an untyped impure functional language. Its standard library provides various primitive data structures (cons cells, tuples, strings, arrays), implemented in the runtime library, but it lacked any higher-level data structures, like finite maps, sets or heaps. We present a toolbox of data structures implemented in Spiral that extend the library with persistent maps, sets and heaps and mutable hash tables. We have also added some new primitives (symbols and mutable references) and fixed a handful of bugs in the runtime.

2 Persistent maps

Persistent maps are located in module `std.map`. The data structure is implemented as an AA tree [2], which is a binary representation of a 2,3-tree, where only right edges can be horizontal. AA trees need to store the rank in every node and are rebalanced using two operations, `skew` (remove horizontal left edges) and `split` (remove two consecutive horizontal right edges).

2.1 Red-black trees

There is also an alternative implementation of persistent maps in module `std.redmap`, employing red-black trees. Very compact functional insertion into red-black trees is presented in [10]. The key idea is that a tree with red root and one red child can be trivially restored to a valid red-black tree by blackening the root. By temporarily allowing this violation of the invariant, we can implement `insert` recursively by propagating the red color up in the tree.

Functional deletion was given 15 years later in [6]. Similar to insertion, deletion is often described as propagating an extra black. We can introduce a third color, black-black, and rebalance the tree in a recursive bottom-up fashion, as we did for insertion. There are more cases to consider, but they are relatively straightforward. This is in contrast to an effective imperative implementation, as described in [5], where the extra black is only conceptual.

3 Persistent sets

Persistent sets, residing in module `std.set`, use weight-balanced binary trees [1]. Our implementation was inspired by the `containers` package in Haskell [9]. The tree is balanced to keep the ratio of the weights of the left and right subtree of every node bounded, so a bottom-up recursive implementation is very natural. One advantage of this balancing scheme is that the sizes of the trees can be used to effectively index into the set.

4 Persistent heaps

Our implementation of persistent heaps from module `std.heap` uses weight-biased variant of leftist heaps described in [10]. The weight-biased leftist property asserts that the size of a right subtree must not be greater than the size of a left subtree, so the right spine of the tree is at most logarithmic in the size of the tree. Heaps can therefore be efficiently merged along their right spines. The main reason for balancing on the weight instead of the rank (length of right spine) is that we have constant time access to the number of elements in the heap.

5 Hash maps

Hash maps are typical data structures that absolutely require mutation, thus cannot be (reasonably) implemented in a pure functional way. Fortunately, impure data structures are as welcome in Spiral as the pure ones. The hash map from module `std.hashmap` uses Robin Hood hashing [4] with linear probing [7, 8], as does `HashMap` from the Rust standard library [11].

The idea of Robin Hood hashing is that upon insertion, the buckets are linearly probed until an empty bucket is found. However, if we find that the inserted entry is further away from its initial bucket than the entry in the probed bucket, we swap them and continue the process with the displaced entry. After removing an entry, we shift the following entries backward until we hit an empty bucket or an entry that resides in its initial bucket (has probe length 0).

5.1 Hashing

Our hashing function is SipHash-2-4 [3], which is claimed to be cryptographically safe. The core algorithm is implemented in the runtime written in C++, because hashing must be fast and requires raw integer operations, none of which is available in Spiral. The Spiral interface to the hasher is exported from `std.hash`.

6 Implementation details

Aside from hashing, we also extended the runtime to support mutable references and symbols. References are mutable boxes that contain exactly one value, which can be used in imperative algorithms as mutable variables (we needed them for hash maps). Symbols are unique objects that can be used as markers or keys in containers. For convenience, symbols wrap a value (usually a string), mostly as an aid for debugging. At the moment, symbols are used to represent color in red-black trees and as markers in data structures (they are represented as tuples, where the first element is the symbol with the name of the structure).

7 Conclusion

We were surprised that even though Spiral is very immature, we found the language quite expressive. The biggest obstacle is the poor error reporting: Spiral itself collects no stack traces, so a C debugger was essential in debugging the programs. Unfortunately, the non-standard calling convention and aggressive inlining performed by the Spiral compiler often made the stack traces unusable, so debugging often resorted to sprinkling the code with `println`s.

Bibliography

[1] Adams, Stephen: *Implementing sets efficiently in a functional language*, 1992.

[2] Andersson, Arne: *Balanced search trees made simple.* In *Algorithms and Data Structures*, pages 60–71. Springer, 1993.

[3] Aumasson, Jean Philippe and Daniel J Bernstein: *SipHash: a fast short-input PRF.* In *Progress in Cryptology – INDOCRYPT 2012*, pages 489–508. Springer, 2012.

[4] Celis, Pedro: *Robin Hood Hashing.* PhD thesis, University of Waterloo, 1986.

[5] Cormen, Thomas H., Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to algorithms.* MIT press Cambridge, 2001.

[6] Germane, Kimball and Matthew Might: *Deletion: The curse of the red-black tree.* Journal of Functional Programming, 24(04):423–433, 2014.

[7] Goossaert, Emmanuel: *Robin Hood hashing.* `http://codecapsule.com/2013/11/11/robin-hood-hashing/`, November 2013.

[8] Goossaert, Emmanuel: *Robin Hood hashing: backward shift deletion.* `http://codecapsule.com/2013/11/17/robin-hood-hashing-backward-shift-deletion/`, November 2013.

[9] *containers: Assorted concrete container types.* `http://hackage.haskell.org/package/containers-0.5.7.1`.

[10] Okasaki, Chris: *Purely functional data structures.* Cambridge University Press, 1999.

[11] *The Rust standard library.* `http://doc.rust-lang.org/std/`.

[12] Špaček, Jan: *Compiler of a simple programming language.* `http://honzasp.github.io/files/spiral_en.pdf`, 2015.