Streams

COMPSCI 220

University of Massachusetts Amherst

Binary Trees with Values at Leaves

The following type and constructors create binary trees where the *values are at the leaves*.

```
1  // type Tree<A> = { kind: 'leaf', value: A }
2  //              | { kind: 'node', left: Tree<A>, right: Tree<A> }
3
4  // leaf<A>(v: A): Tree<A>
5  function leaf(v) {
6      return { kind: 'leaf', value: v };
7  }
8
9  // node<A>(l: Tree<A>, r: Tree<A>): Tree<A>
10 function node(l, r) {
11     return { kind: 'node', left: l, right: r };
12 }
```

The Same-Fringe Problem

The *fringe* of a binary tree are the values in left-to-right order. For example, the fringe of the following tree:

```
1 node(leaf(10), node(leaf(20), leaf(40)))
```

is 10, 20, and 40, in that order. The *same-fringe problem* is to write a function to determine if two trees have the same fringe. For example the following tree has the same fringe as the tree above (but not the same shape):

```
1 node(node(leaf(10), leaf(20)), leaf(40))
```

A Recursive Solution to the Same-Fringe Problem

We can write a simple recursive function to produce the fringe of a tree:

```
1 // fringe<A>(t: Tree<A>): A[]
2 function fringe(t) {
3     if (t.kind === 'leaf') { return [t.value]; }
4     else { return fringe(t.left).concat(fringe(t.right)); }
5 }
```

So, a simple solution to the same-fringe problem is:

```
1 // sameFringe<A>(t1: Tree<A>, t2: Tree<A>): boolean
2 function sameFringe(t1, t2) {
3     return arrayEquals(fringe(t1), fringe(t2));
4 }
```

But, this is not a very efficient solution:

- If the trees are very large, we'll spend a lot of time appending arrays, and
- If the fringes are different, we'll generate the entire fringe instead of terminating as soon as a difference is detected.

Can you think of a better solution?

Binary Tree Iterator

```
1  class TreeIterator {
2      constructor(t) {
3          this.stack = [t];
4      }
5      hasNext() {
6          return this.stack.length > 0;
7      }
8      next() {
9        let top = this.stack.pop();
0        if (top.kind === 'leaf') {
1            return top.value;
2        } else {
3            this.stack.push(top.left);
4            this.stack.push(top.right);
5            return this.next();
6        }
7      }
8  }
```

A typical solution to this problem is to use an iterator. You've seen simple iterators for Java, which iterate over arrays or linked-lists. You can also write an iterator that iterates over the fringe of a tree. The key idea is this: before we descend into the left-hand side of a node to generate its fringe, we need to store the right-hand side of the node in a variable so that we don't forget to generate its fringe. Since the tree may be arbitrarily deep, we may need to remember a stack of nodes.

What would happen if we used a queue of nodes?

An Imperative Solution to the Same-Fringe Problem

```
1  // sameFringe<A>(t1: Tree<A>, t2: Tree<A>): Boolean
2  function sameFringe(t1, t2) {
3      let iter1 = new TreeIterator(t1);
4      let iter2 = new TreeIterator(t2);
5      while (iter1.hasNext() && iter2.hasNext()) {
6          if (iter1.next() !== iter2.next()) {
7              return false;
8          }
9      }
10     return !iter1.next() && !iter2.next();
11 }
12
```

This solution to the same-fringe problem doesn't have the two issues we identified. An unusual feature of this solution is that it uses two iterators simultaneously in a single loop.

What's interesting about this problem is that the solution seems to be fundamentally imperative. In particular, each invocaation of `.next()` returns the next value, which means that the iterators *must* use mutable state internally. Is it possible to solve this problem in a cleaner way?

Memoization

A *memoizer* is an abstraction that "memorizes" the result of a function call.

```
// type Memo<T> = { get: () => T }
// memo1<S, T>(f: (x: S) => T, x: S): Memo<T>
function memo1(f, x) {
  let r = { kind: 'unknown' };
  return {
    get: function() {
      if (r.kind === 'unknown') {
        r = { kind: 'known', v: f(x) }
      }
      return r.v;
    }
  };
}
```

In the following example, we use the memoizer to "memorize" the result of `factorial(10)`. Therefore, the program only calls `factorial` *once*, even though it uses the result twice.

```
function factorial(n) {
    let r = 1;
    while (n > 0) {
        r = r * n;
        console.log("Thinking ...");
    }
    console.log("Done!");
    return r;
}
let f = memo1(factorial, 10);
f.get();
f.get();
```

Note that if we do not call `f.get()`, then `factorial(10)` is not applied. i.e., the memoizer only calls the function if it is *needed*.

Memoization with Thunks

It is straightforward to create memoizers for functions with $2, 3, 4, \cdots$ arguments. The following memoizer is for 0-argument functions (which are called *thunks*):

```
// type Memo<T> = { get: () => T }
// memo0<T>(f: () => T): Memo<T>
function memo0(f) {
    let r = { kind: 'unknown' };
    return {
    get: function() {
        if (r.kind === 'unknown') {
        r = { kind: 'known', v: f() }
        }
        return r.v;
    },
    toString: function() {
        if (r.kind === 'unknown') {
        return '<unevaluated>';
        }
        else {
        return r.v.toString();
        }
    }
    };
}
```

In this memoizer, the `toString` function displays the value returned by `f`, if it has been applied:

```
let f = memo0(function() {
    return factorial(5);
});
console.log(f.toString());
// prints <unevaluated>
f.get();
console.log(f.tostring());
// prints 120
```

Streams

A *stream* is a variant of a linked list, where the tail is memoized:

```
// sempty: Stream<T>
let sempty = {
    kind: 'sempty'
    toString: () => 'sempty'
};

// snode<T>(head: T, tail: Memo: Stream<T>): Stream<T>
function snode(head, tail) {
    return {
        kind: 'snode',
        head: function() { return head; },
        tail: function() { return tail.get(); },
        toString: function() {
            return "snode(" + head.toString() + ", " + tail.toString() + ")";
        }
    }
}
```

An Example Stream

The following is a stream of the numbers $1, 2, 3$:

```
let s1 = snode(1, memo0(() =>
    snode(2, memo0(() =>
        snode(3, memo0(() =>
            sempty))))));

s1.toString();
// returns snode(1, <unevaluated>)
s1.tail().tail().head();
// returns 3
s1.toString();
// returns snode(1, snode(2, snode(3, <unevaluated>)))
s1.tail().tail().tail()
// returns sempty
s1.toString();
// returns snode(1, snode(2, snode(3, sempty)))
```

This is much more verbose than a linked-list! So, if all you need is a linked list, use a linked list. But, there are certain things you can do with streams that you cannot do with linked lists.

Unbounded Streams

The following is a *conceptually infinite* stream of the numbers $1, 2, 3, \cdots$:

```
1 function grow(n) {
2     return snode(n, memo0(() => grow(n + 1)));
3 }
4 let nats = grow(0);
```

Why isn't this an infinite loop?
Notice that `grow` does not directly call itself. Instead, the inner call to `grow` is *guarded* by the thunk.

Higher-Order Functions with Streams

We can define all the usual higher-order functions over streams, for example:

```
// smap<A,B>(f: (x: A) => B, stream: Stream<A>): Stream<B>
function smap(f, stream) {
    return snode(f(stream.head()),
        memo0(() => smap(f, stream.tail())));
}

// sfilter<A>(f: (x: A) => boolean, stream: Stream<A>): Stream<A>
function sfilter(pred, stream) {
    if (pred(stream.head())) {
        return snode(stream.head(),
                     memo0(() => sfilter(pred, stream.tail())));
    } else {
        return sfilter(pred, stream.tail());
    }
}
```

For example, here are two different ways of building the stream of even numbers from the stream of natural numbers:

```
let evens1 = sfilter((x) => x % 2 === 0, nats);
let evens2 = smap((x) => x * 2, nats);
```

Helper Functions to Construct Streams

The following function creates a 1-element stream:

```
// sone<A>(x: A): Stream<A>
function sone(x) {
    return snode(x, memo0(() => sempty));
}
```

The following function appends two streams. However, the second argument is guarded in a thunk:

```
// sappend(left: Stream<A>, right: () => Stream<A>): Stream<A>
function sappend(left, right) {
    if (left === sempty) {
        return right();
    } else {
        return snode(left.head(), memo0(() => sappend(left.tail(), right)));
    }
}
```

Note that sappend behaves in unusual ways when the left-hand stream is unbounded:

```
let evens = sfilter((x) => x % 2 === 0, nats);
let odds = sfilter((x) => x % 2 === 1, nats);
sappend(evens, odds) // is equivalent to evens
```

Fringe as a Stream

Using these functions, we can write a stream that generates the fringe of a binary tree:

```
1 // fringe<A>(t: Tree<A>): Stream<A>
2 function fringeStream(t) {
3     if (t.kind === 'leaf') { return sone(t.value); }
4     else { return sappend(fringe(t.left), () => fringe(t.right)); }
5 }
```

Notice that it is very similar to the function that generates the fringe as an array. However, it behaves quite differently. Since sappend only evaluates its second argument when needed, it doesn't immediately visit all the leaves of the tree, which was the problem with the original function.

Comparing Streams

Since the supplied stream only generate values on need, the `sameStream` function (below) will abort early and produce `false` if (1) it encounters two unequal values or (2) one streams runs out of values before the other. Notice that `sameStream` has the exactly the same structure as the obvious recursive function that checks if two linked-lists are equal.

```
1  // sameStream<T>(s1: Stream<T>, s2: Stream<T>): boolean
2  function sameStream(s1, s2) {
3    if (s1 === sempty && s2 === sempty) {
4        return true;
5    } else if (s1 === sempty || s2 === sempty) {
6        return false;
7    } else {
8        return s1.head() === s2.head() && sameStream(s1.tail(), s2.tail());
9    }
10 }
```

A Stream-based Solution to the Same-Fringe Problem

```
1 // sameFringe<A>(t1: Tree<A>, t2: Tree<A>): boolean
2 function sameFringe(t1, t2) {
3     return sameStream(fringeStream(t1), fringeStream(t2));
4 }
```

Compare this to the inefficient, recursive solution that we started with:

```
1 // sameFringe<A>(t1: Tree<A>, t2: Tree<A>): boolean
2 function sameFringe(t1, t2) {
3     return arrayEquals(fringe(t1), fringe(t2));
4 }
```