Recursion

Structure of the talk

» Recursion

» Corecursion

» Recursion schemes aka. higher level magic

Summing Numbers Recursively

```
// range(0, 1000)
let numbers: number[] = Array.from(Array(1000).keys());

function sumRecur(xs: number[]): number {
  if (xs.length < 1) {
    return 0;
  }

  return xs[0] + sumRecur(xs.slice(1, xs.length));
}

console.log(sumRecur(numbers));
// 499500
```

How does this evaluate?

```
let nums = [0, 1, 2, 3]
sumRecur(nums)
0 + sumRecur([1, 2, 3])
0 + (1 + sumRecur([2, 3]))
0 + (1 + (2 + sumRecur([3])))
0 + (1 + (2 + (3 + sumRecur([]))))
0 + (1 + (2 + (3 + 0)))
0 + (1 + (2 + 3))
0 + (1 + 5)
0 + 6
6
```

Tail Recursion

“Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations”
Wikipedia

TAIL-RECURSIVE SUM

```
function sumTailRecur(nums: number[], acc: number): number {
  if (nums.length < 1) {
    return acc
  }

  return sumTailRecur(nums.slice(1, nums.length), acc + nums[0])
}

console.log(sumTailRecur(numbers, 0));
// 499500
```

TAIL-RECURSIVE SUM

```
function sumTailRecur(xs: number[]): number {
  function sumR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return sumR(nums.slice(1, nums.length), acc + nums[0])
  }

  return sumR(xs, 0);
}

console.log(sumTailRecur(numbers));
// 499500
```

How does this evaluate?

```
let nums = [0, 1, 2, 3]
sumTailRecur(nums)
sumR([0, 1, 2, 3], 0)
sumR([1, 2, 3], 0)
sumR([2, 3], 1)
sumR([3], 3)
sumR([], 6)
6
```

NOTICE THE REPITITION

```
function sumTailRecur(xs: number[]): number {
  function sumR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return sumR(
        nums.slice(1, nums.length),
        acc + nums[0]
    );
  }

  return sumR(
    xs,
    0
  );
}
```

```
function maxTailRecur(xs: number[]): number {
  function maxR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return maxR(
        nums.slice(1, nums.length),
        acc > nums[0] ? acc : nums[0]
    );
  }

  return maxR(
    xs,
    Number.MIN_VALUE
  );
}
```

```
function lengthTailRecur(xs: number[]): number {
  function lengthR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return lengthR(
        nums.slice(1, nums.length),
        acc + 1
    );
  }

  return lengthR(
    xs,
    0
  );
}
```

```
function minTailRecur(xs: number[]): number {
  function maxR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return maxR(
        nums.slice(1, nums.length),
        acc < nums[0] ? acc : nums[0]
    );
  }

  return maxR(
    xs,
    Number.MAX_VALUE
  );
}
```

```typescript
function recurser(
    xs: number[],
    f: (x: number, acc: number) => number,
    base: number
): number {
      function helper(nums: number[], acc: number): number {
        if (nums.length < 1) {
          return acc;
        }

        return helper(nums.slice(1, nums.length), f(acc, nums[0]));
      }

      return helper(xs, base);
}
```

Folds beautiful folds aka. reduce

```
function foldLeft(
    xs: number[],
    f: (acc: number, x: number) => number,
    base: number
): number {
      function helper(nums: number[], acc: number): number {
        if (nums.length < 1) {
          return acc;
        }

        return helper(nums.slice(1, nums.length), f(acc, nums[0]));
      }

      return helper(xs, base);
}
```

John Hughes (1990) Why Functional Programming Matters. Research Topics in Functional Programming.

```
function sumTailRecur(xs: number[]): number {
  function sumR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return sumR(
        nums.slice(1, nums.length),
        acc + nums[0]
    );
  }

  return sumR(
    xs,
    0
  );
}
```

```
function lengthTailRecur(xs: number[]): number {
  function lengthR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return lengthR(
        nums.slice(1, nums.length),
        acc + 1
    );
  }

  return lengthR(
    xs,
    0
    );
}
```

```
function maxTailRecur(xs: number[]): number {
  function maxR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return maxR(
        nums.slice(1, nums.length),
        acc > nums[0] ? acc : nums[0]
    );
  }

  return maxR(
    xs,
    Number.MIN_VALUE
  );
}
```

```
function minTailRecur(xs: number[]): number {
  function maxR(nums: number[], acc: number): number {
    if (nums.length < 1) {
      return acc;
    }

    return maxR(
        nums.slice(1, nums.length),
        acc < nums[0] ? acc : nums[0]
    );
  }

  return maxR(
    xs,
    Number.MAX_VALUE
  );
}
```

MONOIDS!

BEAUTIFUL FOLDS, GABRIEL GONZALEZ, MUNIHAC 2016

Corecursion

IF RECURSION WAS YIN, CORECURSION WOULD BE YANG

Corecursion

“corecursion is a type of operation that is dual to recursion”
Wikipedia

- Recursion starts with data and works (or folds) down towards the base case.
- Corecursion is able use data that it produces from the base case.

Examples

This is where the wheels will probably fall off

Recursion vs. Corecursion Comparisons

https://en.wikipedia.org/wiki/Corecursion#Examples

RECURSION SCHEMES

Meijer E., Fokkinga M., Paterson R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes J. (eds) Functional Programming Languages and Computer Architecture. FPCA 1991. Lecture Notes in Computer Science, vol 523. Springer, Berlin, Heidelberg