L6: Symbolic Processing

CS1101S: Programming Methodology

Martin Henz

September 19, 2018

Subsets: (1) Read, (2) Play

- Given a set *S*, generate all subsets of *S*
- Representation: We represent *S* by a list. Each subset of *S* is also a list, order irrelevant. The result will be a list of subsets (in no particular order), a list of lists.
- Play with example:
  - Given: `const S = list(1, 2, 3);`
  - Wanted:

```
const SS =
    list(list(), list(1), list(2), list(3),
         list(1, 2), list(2, 3), list(1, 3),
         list(1, 2, 3));
```

(3) Think

- How to divide-and-conquer?
  Insight: remember "Coin Change" (L3) and `makeup_amount` (S6)
- Either a specific element is in a subset or not
- The subsets are partitioned by this property
- Append all subsets that include the first element to all subsets that don't

Example

Let $S = \{1, 2, 3\}$.

- Consider first element 1.
- all subsets that **do not** include 1: wishful thinking!
  $S_1 = \{\emptyset, \{2\}, \{3\}, \{2, 3\}\}$
- all subsets that **do** include 1: Just thow 1 into each element of S_1
  $S_2 = \{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$
- Combine S_1 and S_2

(3) Think [a bit harder]

What is the base case? How many subsets does the empty set have?

The empty set has *one* subset.

The set of subsets of the empty set is a set with one element, and that element is the empty set.

(4) Program

```
function subsets(s) {
    if (is_empty_list(s)) {
        return list([]);
    } else {
        const s1 = subsets(tail(s));
        const x = head(x);
        return append(s1,
                      map(ss => pair(x, ss), s1));
}   }

function subsets(s) {
     return accumulate(
         (x, s1) => append(s1,
                           map(ss => pair(x, ss), s1)),
         list([]),
         s);
}
```

Permutations: (1) Read, (2) Play

- Given a set S, generate all permutations of S
- Representation: We represent S by a list. Each permutation of S is a list, with the same elements as S, but possibly in different order. The result will be a list of permutations (in no particular order), a list of lists.
- Play with example:
  - Given: `const S = list(1, 2, 3);`
  - Wanted:

```
const P = list(list(1, 2, 3), list(1, 3, 2),
               list(2, 1, 3), list(2, 3, 1),
               list(3, 1, 2), list(3, 2, 1));
```

(3) Think

- How to divide-and-conquer?
  Insight: Any permutation starts with an element...
- For each element x in S:
  - Generate all permutations of $S - x$
  - Place x in front of each permutation
- Append all results together

Example

Let $S = \{1, 2, 3\}$.

- For each element x:
  - Generate all permutations of $S - x$.
    For example, if $x = 1$, then $S - x = \{2, 3\}$,
    and we have [23], [32]
  - Place x in front of each permutation: [123], [132]
- Same for $x = 2$: [213], [231]
- Same for $x = 3$: [312], [321]
- Append all:
  [[123], [132], [213], [231], [312], [321]]

(3) Think [a bit harder]

- For each element x in S:
  - Generate all permutations of S - x
  - Place x in front of each permutation
- Append all results together

What is the base case? How many permutations does the empty list allow?

The empty list allows for one single permutation, which is also the empty list.

(4) Program

```
function permutations(s) {
    return is_empty_list(s)
        ? list([])
        : accumulate(append, [],
            map(x => map(p => pair(x, p),
                          permutations(remove(x, s))),
                s));
}
```

The problem of sorting

Given

A list `xs` of elements from a given universe X, and a *total order* on X.

Wanted

A permutation of of `xs` such that each element is greater than or equal to the previous one, with respect to the total order.

Comparisons only

The only allowed operations on the elements are comparisons, such as $<$, $>$, $<=$, $>=$, $===$ or $!==$.

How to sort list with n elements?

Our strategy

Wishful thinking! Imagine we can sort lists with m elements, where $m < n$.

Approach A

$m = n - 1$

Approach B

$m = n/2$

Algorithm A1

Idea

Sort the tail of the given list using wishful thinking! *Insert* the head in the right place.

In Source

```
function insertion_sort(xs) {
    return is_empty_list(xs) ? xs
        : insert(head(xs),
            insertion_sort(tail(xs)));
}
```

A1: Insertion Sort

```
function insert(x, xs) {
    return is_empty_list(xs) ? list(x)
        : x <= head(xs) ? pair(x,xs)
            : pair(head(xs), insert(x, tail(xs)));
}
function insertion_sort(xs) {
    return is_empty_list(xs) ? xs
        : insert(head(xs),
            insertion_sort(tail(xs)));
}
```

Complexity of Insertion Sort?

```
function insert(x, xs) {
    return is_empty_list(xs) ? list(x)
        : x <= head(xs) ? pair(x,xs)
            : pair(head(xs), insert(x, tail(xs)));
}
function insertion_sort(xs) {
    return is_empty_list(xs) ? xs
        : insert(head(xs),
            insertion_sort(tail(xs)));
}
```

Algorithm A2

Idea

Find the smallest element x and remove it from the list. Sort the remaining list, and put x in front.

A2: Selection Sort in Source

```
function selection_sort(xs) {
    if (is_empty_list(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(x,
                    selection_sort(remove(x, xs)));
    }
}
```

A2: Selection Sort (function `smallest`)

```
// find smallest element of a non-empty list xs
function smallest(xs) {
    function sm(ys, x) {
        return is_empty_list(ys) ? x
            : x < head(ys) ? sm(tail(ys), x)
                : sm(tail(ys), head(ys));
    }
    return sm(tail(xs), head(xs));
}
```

Complexity of Selection Sort?

```
function selection_sort(xs) {
    if (is_empty_list(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(s,
                    selection_sort(remove(x, xs)));
    }
}
```

Recall: How to sort list with n elements?

Our strategy

Wishful thinking! Imagine we can sort lists with m elements, where $m < n$.

Approach A

$m = n - 1$

Approach B

$m = n/2$

Idea of Algorithm B1

Split the list **in half**, sort each half using wishful thinking, **merge** the sorted lists together

B1: Merge Sort

```
function merge_sort(xs) {
    if (is_empty_list(xs) ||
        is_empty_list(tail(xs))) {
        return xs;
    } else {
        const mid = middle(length(xs));
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs, mid)));
    }
}
```

B1: Merge Sort (function `merge`)

```
function merge(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else if (is_empty_list(ys)) {
        return xs;
    } else {
        const x = head(xs);
        const y = head(ys);
        return (x < y)
               ? pair(x, merge(tail(xs), ys))
               : pair(y, merge(xs, tail(ys)));
    }
}
```

Helper functions for Merge Sort

```
// take half, rounded down
function middle(n) {
    // Reflection R6
}
// put the first n elements of xs into a list
function take(xs, n) {
    // Reflection R6
}
// drop first n elements from list, return rest
function drop(xs, n) {
    // Reflection R6
}
```

Complexity of Merge Sort?

```
function merge_sort(xs) {
    if (is_empty_list(xs) ||
        is_empty_list(tail(xs))) {
        return xs;
    } else {
        const mid = middle(length(xs));
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs, mid)));
    }
}
```

Algorithm B2

wait for Mission “Sorting Things Out”

Representing functions: Directly

Our first approach is to represent functions *directly* in Source.

Example

```
function my_f(x) {
    return x * x + 1;
}
function eval_numeric(f, x) {
    return f(x);
}
eval_numeric(my_f, 7); // returns 50
```

Describing the graph of functions

```
// make a graph curve for function f;
// the graph covers the range for x from x1 to x2
function function_to_graph(f, x1, x2) {
    function graph(t) {
        // for t from 0 to 1,
        // x ranges from x1 to x2
        const x = x1 + t * (x2 - x1);
        return make_point(x, f(x));
    }
    return graph;
}
```

Plotting the graph of functions

```
// plot the graph of function f from x1 to x2
function plot_graph(f, x1, x2) {
    return (draw_connected_squeezed_to_window(200))
           (function_to_graph(f,x1,x2));
}
```

Numerical Differentiation

```
// numerical differentiation; simplest method
const dx = 0.0001;
function deriv_numeric(f) {
    return x => (f(x + dx) - f(x)) / dx;
}
```

Symbolic evaluation

Now we represent functions with data structures!

Example expression

```
// my_exp represents x * x + x + 4
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", make_number(1)));
```

Symbolic evaluation

```
eval_symbolic(my_exp, "x", 3);
// should return 16
```

Symbolic differentiation

Example expression

```
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", make_number(4)));
```

Symbolic differentiation

```
deriv_symbolic(my_exp, "x");  // should return
// make_sum(make_product("x", make_number(2)),
//          make_number(1))
eval_symbolic(deriv_symbolic(my_exp,"x"),"x",3);
// should return 7
```

Symbolic representation of functions

- `is_variable(e)`: Is `e` a variable?
- `is_same_var(v1, v2)`: Are `v1` and `v2` same variable?
- `is_sum(e)`: Is `e` a sum?
- `addend(e)`: Addend of the sum `e`
- `augend(e)`: Augend of the sum `e`
- `make_sum(a1, a2)`: Construct the sum of `a1` and `a2`
- `is_product(e)`: Is `e` a product?
- `multiplier(e)`: Multiplier of the product `e`
- `multiplicand(e)`: Multiplicand of the product `e`
- `make_product(m1,m2)`: Construct product of `m1` and `m2`

Implementing `eval_symbolic`

```
function eval_symbolic(exp, name, val) {
 return is_number(exp) ? value(exp)
  : is_variable(exp) ?
     (is_same_var(exp, variable) ? val : NaN)
  : is_sum(exp) ?
     eval_symbolic(addend(exp), name, val) +
     eval_symbolic(augend(exp), name, val)
  : is_prod(exp) ?
     eval_symbolic(multiplier(exp), name, val) *
     eval_symbolic(multiplicand(exp), name, val)
  : "unknown exp type: " + exp;
}
eval_symbolic(square, "x", 4);
```

Symbolic differentiation

The rules

$$
\begin{aligned}
\frac{dc}{dx} &= 0 \qquad \text{for constant } c \\
\frac{dx}{dx} &= 1 \\
\frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx} \\
\frac{d(uv)}{dx} &= u\frac{dv}{dx} + v\frac{du}{dx}
\end{aligned}
$$

Definition of `deriv_symbolic`

```
function deriv_symbolic(exp,v) {
  return is_number(exp) ? make_number(0)
    : is_variable(exp)
      ? make_number(is_same_var(exp, v) ? 1 : 0)
    : is_sum(exp)
      ? make_sum(deriv_symbolic(addend(exp), v),
                 deriv_symbolic(augend(exp), v))
    : is_product(exp)
      ? make_sum(make_prod(multiplier(exp),
            deriv_symbolic(multiplicand(exp),v)),
          make_prod(multiplicand(exp),
            deriv_symbolic(multiplier(exp), v)))
    : "unknown exp type: " + exp;
}
```

Revisiting example

Example expression

```
const my_exp = make_sum(make_product("x", "x"),
                        make_sum("x", make_number(4)));
```

Symbolic differentiation

```
deriv_symbolic(my_exp, "x");  // should return
// make_sum(make_product("x", make_number(2)),
            make_number(1))

// but instead returns complicated expression
// equivalent to: x * 1 + x * 1 + 1 + 0
```

Simplifying formulas: `make_sum`

```
function make_sum(a1,a2) {
  return is_number_equal(a1, 0) ? a2
       : is_number_equal(a2, 0) ? a1
       : is_number(a1) && is_number(a2)? a1 + a2
       : list("+", a1, a2);
}
```

Simplifying formulas: make_product

```
function make_product(m1, m2) {
  return is_number_equal(m1,0) ||
         is_number_equal(m2,0)              ? 0
       : is_number_equal(m1,1)              ? m2
       : is_number_equal(m2,1)              ? m1
       : is_number(m1) && is_number(m2) ? m1 * m2
       : list("*",m1,m2);
}  }
```

Symbolic differentiation: Constructors

```
function make_sum(exp1, exp2) {
    // for example as a list
    return list("+", exp1, exp2);
}
function make_prod(exp1, exp2) {
    return list("*", exp1, exp2);
}
```

Symbolic differentiation: Accessors

```
function multiplier(exp) {
    return head(tail(exp));
}
function multiplicand(exp) {
    return head(tail(tail(exp)));
}
```

Symbolic differentiation: Predicates

```
function is_sum(x) {
    return is_pair(x) && head(x) === "+";
}

function is_product(x) {
    return is_pair(x) && head(x) === "*";
}
```

Process of designing data structures

- Specification: describes *what* is done (contract)
- Implementation: describes *how* it is done (work)

Sets: Specification

A set is an *unordered* collection of objects.
Let us consider as objects only *numbers*, here.

- Constructors: `make_empty()`, `add(e,s)`, `remove(e,s)`
- Predicates: `find(e, s)`
- Contract:
  - `find(e, add(e, s))` returns `true`
  - `find(e, remove(e, s))` returns `false`

Set: Implementation (version 1)

Idea: represent sets by *unordered* lists

- Consequence for `add(e, s)`?
- Consequence for `remove(e, s)`?
- Consequence for `find(e, s)`?

Set: Implementation (version 2)

Idea: represent sets by *ordered* lists

- Consequence for `add(e, s)`?
- Consequence for `remove(e, s)`?
- Consequence for `find(e, s)`?

Set: Implementation (version 3, Mission “S & R”)

Idea

represent sets by *binary search trees*

Binary search tree property

Every number in the left tree is smaller than the number, and every number in the right tree is larger than the number

- Consequence for `add(e, s)`?
- Consequence for `remove(e, s)`?
- Consequence for `find(e, s)`?

Which implementation of Set is the best?

Several reasonable choices

- Some operations are fast in one implementation and slow in another one
- Often there is a trade-off
- *Best* representation depends on which operations are most frequent.

Key Ideas

- Challenges from S6: subsets and permutations
- Sorting (quadratic and $\Theta(n \log n)$)
- Differentiation (numeric and symbolic)
- Data structure design: consider implementation trade-offs